2026 ASEE ANNUAL CONFERENCE June 21–24, 2026 | Charlotte Convention Center, NC
Host Sponsor UNC Charlotte
ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION
Paper ID #49985

Antagonistic Development via Intentional Software Churn

Andrew James Goodling, Grand Valley State University
Cameron John Schneider
Dr. Erik Fredericks, Grand Valley State University

Erik Fredericks is an Associate Professor in the College of Computing at Grand Valley State University. His research focuses on exploring how uncertainty can impact self-adaptive and safety-critical systems at different levels of abstraction and how it can be mitigated by using search-based software engineering techniques. Recently, he has been investigating how generative art can be automatically created via evolutionary computation.

Reihaneh Hariri, Grand Valley State University

Antagonistic Development via Intentional Software Churn

Abstract

Software churn is a byproduct of the software development process where frequent updates often lead to technical debt, where technical debt is the act of leaving issues, efforts, or refactoring for later and can easily result in a significant amount of time spent resolving those issues. However, when development teams are creating separate features in parallel then intentional churn can drive necessary updates between teams, thereby leading to a more useful and resilient end product especially when not all requirements have been sufficiently-defined prior to the development cycle. This paper focuses on the interplay between two project features, each developed independently, where intentional updates to one feature drive changes in the other feature.

This project was constructed as a research collaboration between two Master's students for their final project with the intent of creating two separate features for a simulated drone application in a short amount of time. The first aspect of the project was the development of a resilient, autonomous drone controller that would successfully navigate to a specified location in uncertain scenarios. The second aspect was to create a framework for procedurally creating environmental scenarios that the drone must successfully navigate. Development was split into two separate teams with antagonistic goals towards each other: robot control and procedural content generation. Each team's overarching goal was to "break" the other's system via forward progress with their own goal(s), where progress was tracked in public open-source repositories. This paper reports on an educational experience involving intentional software churn to aid development in a collaborative software engineering project.

1 Introduction

The phrase "commit early, commit often" is often used when using git-style source control, where the intention implies that incremental fixes are tracked over time. However, frequent updates to revision control can result in software churn, where churn has the potential for poor software quality [1–6]. In a DevOps environment, churn can also provide feature-oriented teams with information about progress made by others [7]. For example, a team can be automatically notified if another team updates project artifacts, thereby enabling localized updates if necessary. Depending on the domain, such an approach can also support "antagonistic development" – teams that strive to break each others' systems may lead to a more resilient system than previously considered. Such an approach may highlight misunderstood or incomplete requirements, uncertainty in initial design decisions, and unknown yet required features.

Development processes range from highly-structured (e.g., Waterfall) to flexible and fast-paced (e.g., Agile) and deployment of those techniques is typically domain- and/or experience-dependent [8]. Experimental development processes, such as parallel development [9–11], aim to break the mold of tradition and pit teams against each other in the workplace on developing common or identical features. However, such processes can be detrimental to overall development and morale if not managed appropriately. Co-evolution describes how updates in one artifact may drive changes in another [12]. Additionally, software metrics such as code churn, commit logs, and burn-down/burn-up charts can provide insights into a team's progress over time. We focus on a variation of parallel development and software co-evolution where teams develop separate aspects of an overall project with the goal of improving each others' code and artifacts via continuous updates.

This paper describes a proof of concept deployment of antagonistic development that was supported with automated notification systems from open-source revision control. Two teams were tasked with developing individual modules with limited requirements to deliver a functioning system at the end of a 15 week period. One team was required to develop an autonomous control system for a drone and the other team was required to design a procedurally-generated set of environments that would demonstrate the capabilities of the drone controller. Each team worked individually with the intent of overcoming uncertainties and/or adversities designed by the other team.

For this paper the teams comprised Masters students working on their final deliverable to complete a Masters Project (i.e., required for graduation at our university). Over the course of the project each team made significant progress in their efforts, delivering a more robust and complete project than could have been performed individually.[1] Together, the delivered controller and procedural generation elements yielded a complete experimentation framework for future research projects. Additionally, the resulting data parsed from monitoring our source control repository demonstrates reciprocal commits that are in response to an added feature or update.

The remainder of this paper is organized as follows. Section 2 presents background information on automated software development, drones, and procedural content generation (PCG). Section 3 details our approach for implementing antagonistic development in the context of an autonomous

[1] This conclusion is the result of experience from a large number of past projects of similar needs.

drone system. Section 4 presents our experimental results and Section 5 discusses related work. Lastly, Section 6 discusses our efforts and presents future avenues for this path of research.

2 Background

This section presents background information on automated software development processes, autonomous drones, and PCG.

2.1 Automated Software Development Processes

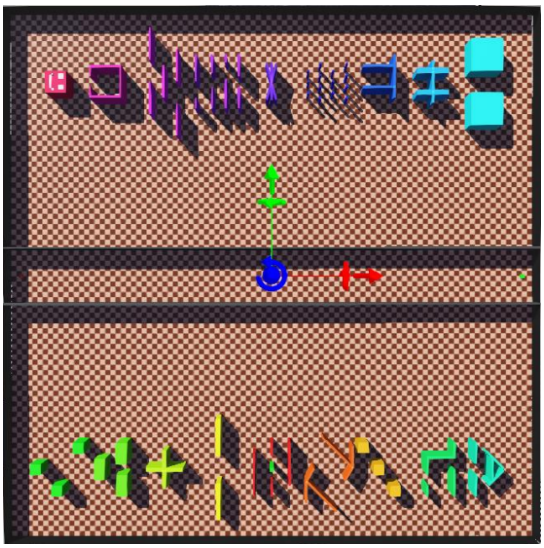
Software development processes are structured approaches for ensuring that delivered software artifacts follow a standard procedure for quality and repeatability [13, 14]. The rigidity of the process is dependent on the followed process. For example, a Waterfall or V-model approach offers little flexibility with respect to changing requirements, whereas more fluid processes such as Agile enable quick responses to rapid changes in requirements or key objectives [15]. While the individual phases, or high-level activities, of a software project have been long defined [16], various software processes differ substantially on how and when the work within those phases is performed. Regardless, a software development process must be followed to ensure that all stakeholders in a project are able to successfully work together to deliver a quality product. Importantly, software process models of all variations are intended to enhance successful collaboration by minimizing contention within the software process. Both Waterfall and the V-model reduce in-development contention with the customer at the cost of delayed customer feedback while Agile methodologies reduce contention in delivery due to higher customer involvement during development. For the purposes of this paper, we follow an Agile style approach where sprints are guided by weekly standup meetings and change is embraced.

DevOps is a process in which automated tools are used to support a software development lifecycle to enhance collaboration and deliverability of projects [17]. Common DevOps activities include automated build and test cycles, automated notifications of status changes in source repositories, and project management activities (e.g., issue tracking, milestones, task assignments, etc.) in an online/remote environment. For example, a team can push changes to a remote repository and receive an automated report concerning build issues, test reports, and integration activities. Online platforms such as GitHub and Mercurial provide software teams with the ability to use DevOps activities without local installation. We used GitHub for this project given the authors' experience with the platform.

2.2 Autonomous Drones

Drones (i.e., unmanned autonomous vehicles (UAVs)) are typically considered to be safety-critical systems where failure can be catastrophic [18]. Such systems can be operated manually via a remote human pilot or autonomously via control strategies. Drones are often equipped with a number of rotors, sensors, and components that enable flight activities and are typically tasked with mission objectives that include photography, monitoring, and data collection. For the purposes of this paper, we use a simulation of a Bitcraze Crazyflie 2.1 micro-drone [19] in the Webots simulator [20]. This particular drone comprises four rotors and

(a) *Team 1*: Crazyflie drone in empty four-walled simulation world.



(b) *Team 2*: Procedurally-placed obstacles in simulation world.

Figure 1: Screenshots from *Team 1* and *Team 2* simulation environments.

uses a multi-ranger deck for object detection and an optical flow deck for measuring distance to the ground. Figure 1(a) shows a screenshot of our drone in the Webots simulation environment. There are a large number of strategies for autonomous control of drones, including rules-based, reactive (i.e., responding to sensor values), and intelligent approaches (e.g., neural networks, large language models, etc.) [21, 22].

Motivating Example. Our mission objective was for a drone to autonomously traverse a procedurally-generated, maze-like environment and successfully reach a waypoint without colliding with an obstacle/wall or "timing out" by spending too long in a room and/or the simulation as a whole. Our control strategy was reactive – the drone polled its obstacle detection sensors to determine if there was clearance to move forward. If not, the controller would attempt to follow a wall and rotate as necessary to find clearance to continue its forward progress. The simulation environment is shown in Figure 1(b). The drone would be instantiated at the left of the environment within the narrow hallway and be tasked to fly to the rightmost wall, where its destination is represented by a green cube. The obstacles shown in Figure 1(b) would be placed within the hallway as part of the PCG algorithm (next described) prior to the drone launching.

2.3 Procedural Content Generation

PCG is an approach for generating and/or placing content in a space using algorithms or heuristics, with the intent being to relieve the developer of the burden of manually creating a large number of varied environments [23, 24]. PCG is often used in the video game and simulation domains to increase the amount of content available to experience, from generating environments to procedurally-animating entities in a scene. Typically, PCG is enabled via mapping computational noise functions to entities in a scene [25] or by using algorithms to generate

environments (e.g., cellular automata [26]). However, without strict requirements PCG can lead to outcomes that are infeasible or unsatisfactory. To alleviate this problem some applications use a combination of hand-crafted environments that are procedurally-placed in a scene, most notably in the video game The Binding of Isaac.[2] In this game, the designers hand-crafted a large number of rooms and used procedural generation techniques for placing them in an environment, thus balancing a tradeoff between manual effort and random generation. We followed a similar approach in that a number of rooms were manually created in the Webots simulation environment and procedurally-placed and managed in a scene.

Motivating Example. Initially a number of rooms were created in the Webots environment with the intention of providing varied obstacles for the drone to manage, where a subset of those pictured in Figure 1(b) were initially developed. The initial goals were to create a number of rooms that ranged from very simple (i.e., a single block) to very complex (i.e., including dead ends and maze-like features). The drone had to traverse a maximum of nine rooms in sequence before reaching its waypoint objective, starting in an empty room (i.e., `START`) and ending in an empty room with a green cube representing the waypoint (i.e., `END`)). Rooms in between `START` and `END` were selected from the hand-crafted room types seen on the sides in Figure 1(b). At present, room placement is completely random. Additional room types were required to offset updates made as development of the controller progressed. Rooms with tight corridors and angled walls were added for additional challenge. Once the drone controller was able to solve these rooms, moveable obstacles were added to further enhance the challenge and prior builds were refined to add extra difficulty. In total, 18 different types of rooms were added to the simulation.

3 Approach

This section describes our approach. To motivate antagonistic development, consider this quote from one of the participants: "How can I make Author #1's life the worst this weekend?"[3]

3.1 Assumptions, Inputs, and Outputs

We now discuss our assumptions, required inputs, and expected outputs for this project. We assume there are at minimum two project teams capable of working independently. For each project team, a necessary amount of software artifacts are required to be in place to enable development (e.g., requirements, user stories, test cases). The artifacts must at minimum specify a base level of functionality and must be flexible to accommodate frequent updates. Additionally, we assume that each team works on an independent module or set of features that are distinct from those work packages of the other teams (i.e., this is not intended to be duplicate effort). We also assume that the software teams leverage an automated build pipeline that at minimum automatically notifies each team that an update has been pushed and built. Expected outputs include released software builds, updated software documentation artifacts, and updated traceability links between all artifacts.

[2] See `https://store.steampowered.com/app/250900/The_Binding_of_Isaac_Rebirth/`.

[3] This was said in jest however was a factor in driving forward progress.

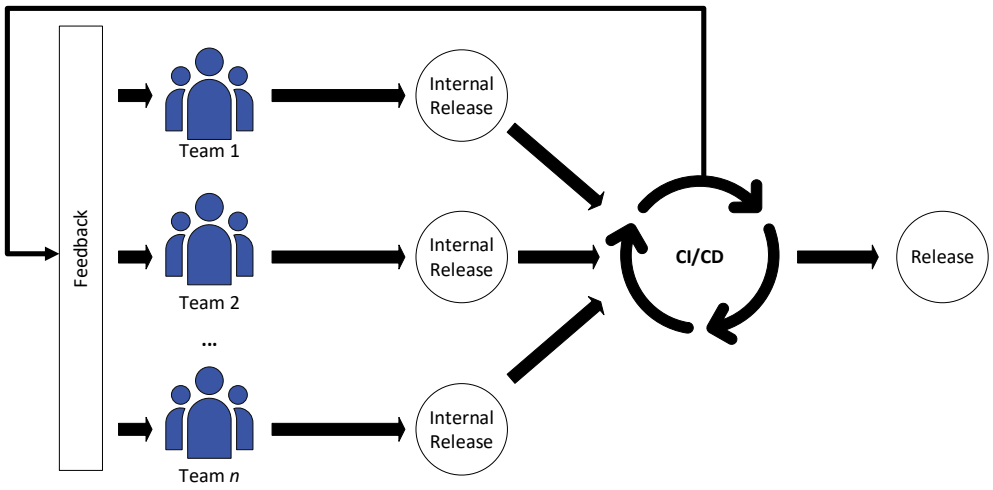


Figure 2: Project workflow with antagonistic development procedure.

3.2 Antagonistic Development

At its core antagonistic development resembles a traditional software development lifecycle, where testing supplements development by providing feedback on adherence to requirements. In our approach each team's internal release candidate is pushed to a CI/CD pipeline and other teams are automatically notified of the changes and potential impacts to their work packages. Software churn is embraced to drive intentional updates that require forward progress from other teams to mitigate the changes made.

Figure 2 presents a workflow of the intended process. Specifically, each team works in parallel on their respective features/modules. Upon completion of a task or sprint, a team will push their changes to a CI/CD pipeline and all teams are notified of the push with documentation of all changes. When ready, each team whose modules are affected by the pushed change will pull down changes and perform any necessary integrations. The updated modules are then tested to determine what changes must be incorporated as a result of the initial team's updates.

Note that this process, if not properly managed, can lead to a negative development environment where personnel issues may arise. It is therefore critical to ensure all participants understand the nature of the development process and that all work tasks are aimed at improving the overall application, rather than aiming to intentionally "sink" another team with overly-frequent updates.

Motivating example. To illustrate this process we describe our implementation of antagonistic development in an academic setting. For this project there were two teams (*Team 1* and *Team 2*) that were each responsible for two separate modules within our drone simulation, where each team comprised one graduate student.[4] *Team 1* was responsible for developing an autonomous drone controller that would navigate uncertain environments, with the goal of reaching a target waypoint without being damaged. *Team 2* was responsible for developing a procedurally-generated set of environments that were required to be navigable yet challenging for

[4]Each graduate student performed this work as part of their respective Masters projects.

an autonomous drone. Both teams followed an Agile approach to development comprising weekly development sprints and weekly standups to discuss progress and key objectives. All software artifacts from both teams were housed within a GitHub repository with each team maintaining their own separate branch. The repository was configured to automatically notify collaborators when changes were pushed.

Requirements for this project were intentionally sparse at the outset, focusing on two key objectives for the drone: **(1) Avoid collisions** and **(2) Navigate to waypoint**. Essentially, if the drone collided with anything or failed to reach its intended waypoint then the simulation was considered a failure.

Figure 1 shows screenshots from the Webots simulation environment used by our teams and will be further described in our sample scenarios. Specifically, Figure 1(a) shows the Crazyflie drone used in the simulation, where in this image the drone is located within an empty four-walled environment and is performing a "wall following" task. Here, the drone flies until it discovers a wall and then maintains a configurable distance from that wall and flies in parallel. Upon discovering another wall, the drone will change direction and follow the new wall. Figure 1(b) shows a top-view of the simulation world where a number of hand-crafted obstacles are placed out of view of the "main" corridor that the drone must navigate. Obstacles are placed along the corridor based on the procedural placement algorithm and the drone will need to navigate to the right-most edge of the world.

Next, we describe three different scenarios to illustrate antagonistic development. Note that these scenarios are meant to motivate its potential via our observations during this process – there exist many other scenarios that merit further exploration as part of our future work.

Scenario 1: Complementary Development. In this scenario both teams are working separately on individual tasks that induce positive changes in other team's modules. For example, *Team 1* develops an initial drone controller in a sample simulation world (i.e., a default world solely comprising four walls) to demonstrate proof of concept (c.f., Figure 1(a)). This controller involves the drone performing a "wall-following" task with little flexibility for unexpected obstacles. *Team 2* develops a separate simulation world in the simulator to understand how to procedurally-place collideable obstacles within the scene. Specifically, rectangular prisms of varying length, height, and depth are placed randomly within the same default walled world (c.f., Figure 1(b)). When *Team 2* makes a GitHub push then *Team 1* is notified and pulls down the simulation world, testing their new drone controller in the environment. *Team 1* quickly learns about shortcomings in the current version of the drone controller (i.e., via unexpected collisions with the procedurally-generated obstacles) and implements a set of changes that provide a larger tolerance to avoid the new obstacles. Once *Team 1* is satisfied they push new changes to GitHub, notifying *Team 2* and continuing the cycle.

This process appears to be an "arms race" and can quickly get out of control unless tasks are well-defined by a project lead and moreover that critical paths of communication are kept open between all involved. Note that at this point, system requirements may be in an early development phase or may be relatively complete, leading to development focusing on satisfying our key objectives. The specific requirements supporting those objectives are intended to be updated as a result of the antagonistic development process. For example, at this stage it was determined that

Team 1 needed additional requirements that specify how sensors are monitored and how the motors are controlled. Additionally it was found that controller logic was very brittle with respect to unforeseen environmental scenarios. Once the changes were made to all of *Team 1*'s artifacts (i.e., code and documentation), *Team 2* was notified via a push and continued iterating their module to further exercise the autonomous controller. In this case, both teams were working in parallel and updates from each further clarified unclear/missing requirements.

Scenario 2: Deleterious Push. In this scenario one team makes a push that sets back progress on another team, thereby requiring significant revision and/or rethinking of existing artifacts. For example, *Team 2* developed a sequence of obstacles that would require precise navigation from the drone with very little clearance for movement (c.f., Figures 7 and 8). However, the drone's controller specified buffers that enables the drone to reorient itself with minimal risk of collision (i.e., $0.3m$ for obstacle avoidance, slowdown threshold of $0.7m$, etc.). This buffer was too large for the gaps, leading to a situation in which the drone's controller would not allow forward motion even though the drone physically could have fit. This particular issue led to a number of timeout errors as the drone essentially became "stuck" within the room and drove a redesign of *Team 1*'s buffer strategy.

Scenario 3: Parallel Development. In this scenario both teams are working separately on individual tasks that, in theory, have no direct impact on the other (i.e., no code-breaking changes, such as updated function signatures). This might be thought of as a more typical development cycle where individual teams focus on their core responsibilities prior to integration activities. For example, *Team 1* was developing a "quick fly" feature that would enable the drone to speed up when moving walls are detected. *Team 2* was refining their moving obstacle feature to provide additional challenge for the drone. When *Team 1* pushed their code up to the repository, *Team 2* was able to pull the update and test out the new controller on their revised obstacle motion code. In this scenario, neither update necessitated an immediate change from either team, however changes were able to be used for localized testing activities prior to feature integration.

4 Experimental Results

This section presents our experimental configurations and results, respectively, for co-development of an autonomous drone system in procedurally-generated environments. We first discuss our adherence to the antagonistic development approach, then describe our experimental configuration for the software experiment, and lastly present our experimental results.

4.1 Antagonistic Development Implementation

We previously described our project and application of antagonistic development in Section 3.2.

All controllers and simulators were developed with Webots R2032b in both C and Python, depending on the module.[5] Code and simulator configurations were continuously pushed to individual branches on GitHub and each developer was notified when a push occurred. Once a release was ready to be tested, both teams would discuss changes and cooperatively test their

[5] See `https://cyberbotics.com/`.

modules with the current state of their builds. Following, a list of required changes and updates would be developed to determine the requirements of the next sprint.

4.2 Experimental Configuration

We performed our experiments on a PC running Windows 10 with a Ryzen 7 5800x CPU, an Nvidia 3070Ti FE GPU, and 32GB of DDR4 RAM. For this paper we configured a controller in Webots to automatically reset the simulation when the drone either reached a `FAIL` state or if it successfully reached its objective, where a `FAIL` state was either a result of a collision or if the drone was in a room or in the simulation for an extended period of time without reaching its objective. The room timeout was set to 60 seconds and the global simulation timeout was set to 60 seconds $*$ *the number of rooms* + 60 seconds. For example, if there were 7 rooms then the global timeout would be 8 minutes.

Each time the simulation reset a new random and unique seed value was selected and stored for reproducibility. The overall simulation was run continuously for 10 hours resulting in 7620 experimental evaluations. Our anonymized GitHub repository is available here, however all author information and activity has been removed for the purposes of this submission: `https://anonymous.4open.science/r/AnonConfRepo-8838/`.[6]

4.3 Experimental Results

This project had three overarching goals: demonstrate the feasibility of antagonistic, develop an autonomous drone controller, and develop a strategy for procedural environmental generation to demonstrate the capabilities of the drone controller.

4.3.1 Antagonistic Development

This project was performed over a 15 week semester as part of a Master's Project sequence where two students worked together on common objectives. One student comprising *Team 1* was tasked with developing the autonomous drone controller and the other student comprising *Team 2* was tasked with developing the technique for procedurally-generating environments for the drone. At the outset of the project it was agreed upon that the teams would follow a strategy that would significantly impact the other's efforts as part of an approach for continuous improvement (i.e., antagonistic development).

For each week of the semester a collaborative meeting was held between both teams and the team lead (i.e., the Masters Project advisor) to discuss recent updates, plan for the following week's efforts, and demonstrate progress. Both teams were given a set of key objectives to fulfill, with specific requirements left intentionally vague to enable freedom of choice with respect to development. Additionally, all members of this project were in communication via a messaging service that enabled asynchronous discussions in the event that a significant source push occurred outside of the scheduled meeting time.

[6]Note that this anonymous repository has removed all branches and commit logs – these will be available following the review process.

Table 1 presents an overview of the activities that occurred in the project GitHub repository. Of note there was not a formal strategy for managing the GitHub repository – team members were given the freedom to create branches to work on individual features and there were no restrictions on push activities.

Total number of commits (including branches):	142
Number of branches:	5
Number of pull requests:	3

Table 1: Overview of GitHub activities.

Figure 3 visualizes the commit history between both teams over the duration of the project. Of note is that typically one team would make a commit and then the other team would commit changes shortly thereafter. For example, *Team 2* made a change that added a new room with diagonal corridors and *Team 1* immediately updated their controller to account for motor speed control. We also note that the the majority of commits are grouped - there are very few singular commits without a "response" commit.

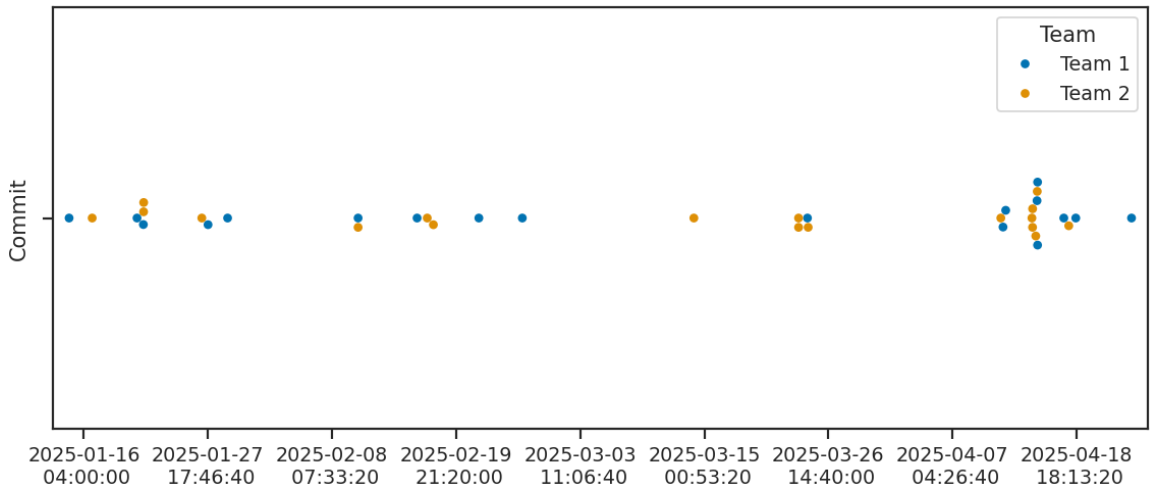


Figure 3: Comparison of GitHub commits between both teams.

This goal is difficult to empirically quantify without a comparison to other development processes and we leave that for future studies. However, from past experience this development strategy was a significant improvement to a more traditional process and resulted in both robot controllers and environment generation approaches that far outpaced other projects performed by the authors and their respective research labs.

4.3.2 Autonomous Drone Controller and Procedural Environment Generation

Table 2 summarizes our overall run data, comprising the overall number of successful runs and failures. Of note, the number of collisions significantly outweighs all other metrics. These results imply that our controller, while having some successes, still requires additional iteration to handle the environmental uncertainties generated via our PCG controller. However, the drone's progress

Total number of runs:	7620
Overall simulation time:	10 hours
Successful runs:	778 (10%)
Failed runs (collision):	5594 (73%)
Failed runs (room timeout):	599 (8%)
Failed runs (global timeout):	649 (9%)

Table 2: Summary of experimental results from Webots simulation.

from early builds did significantly improve throughout the course of development. For example, one push from *Team 2* introduced moveable obstacles where objects in the environment would follow a pattern during the simulation. The sensors available on the drone within the simulation cannot perform full 360 degree sensing and had "blind spots."[7] As such, updates from *Team 1* to mitigate this issue included spinning the drone while moving to detect if new obstacles appeared and enabling "speed boosts" to accelerate the drone through openings in the environment.

Next, we analyze the specific rooms that resulted in success or failure. First, we look at the ordering of the rooms. As mentioned in Section 2.3 each individual room was manually created in the Webots environment and then placed in the sequence of rooms. Table 3 shows the number of failures based on the room order. As can be seen here, the largest number of failures occurs in the first procedurally-generated room of the sequence – where `START` is an empty room to initialize the drone. Of interest is that the number of failures decreases as the drone proceeds through rooms, indicating that the drone tends to fail earlier in the sequence.

Room sequence number	**Number of failures**
START	1065
1	3166
2	1573
3	606
4	260
5	130
6	42
7	0
8	0
9	0
END	0

Table 3: Number of failures based on the room ordering.

Next we examine the different types of failures on a per-room basis to understand which types of rooms induced problems. Figures 4, 5, and 6 show the total number of failures resulting from collisions, room timeouts, and global timeouts for each specific room, respectively.

[7] Interestingly, the physical drone does not have this same limitation.

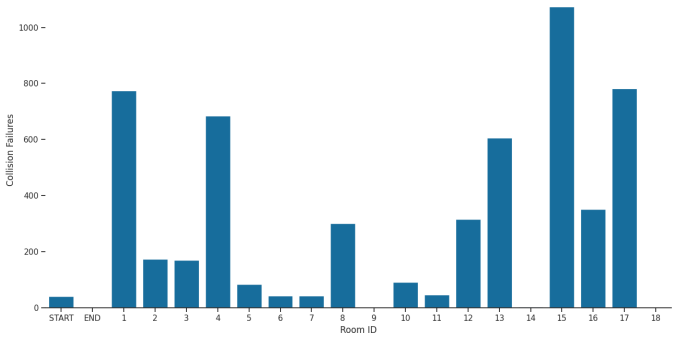


Figure 4: Total number of collisions by room.

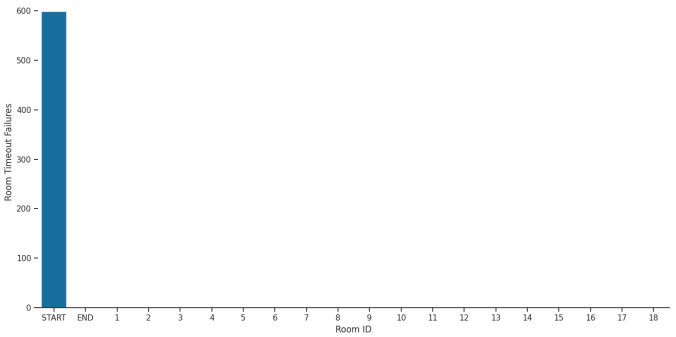


Figure 5: Total number of room timeout failures by room.

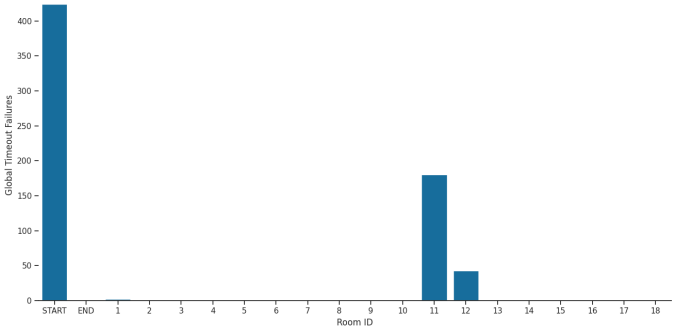


Figure 6: Total number of global timeout failures by room.

As shown in Figure 5, the `START` room is the only room to induce a room timeout error, indicating that the drone controller simply got stuck in an open space and could not recover. Interestingly, no other room configurations led to this error, indicating that the drone either was able to escape the room, collided with an obstacle, or experienced a global timeout before experiencing the localized timeout. Figure 7 shows a screenshot of Room ID 1 and Figure 8 shows a screenshot of Room ID 15, both of which correspond to the largest number of collisions. Each of these rooms comprises a number of relatively thin rectangular prisms with little clearance that caused the drone to either fly in circles or incidentally clip a wall when passing through an opening.

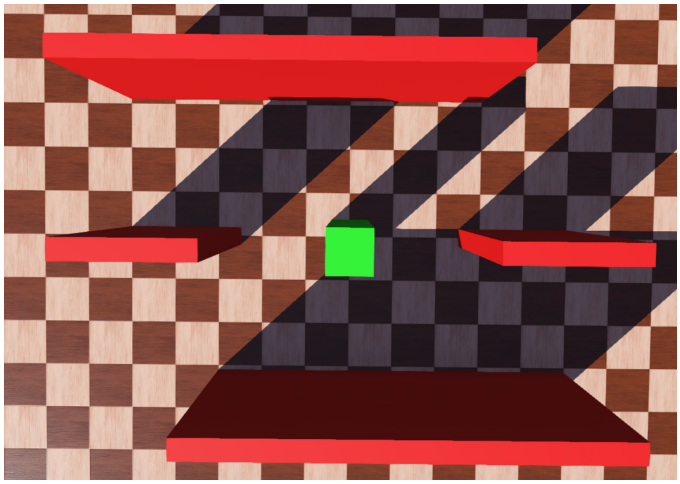


Figure 7: Screenshot of Room ID 1.

Figure 8: Screenshot of Room ID 15.

Conversely, Figure 6 shows the number of global timeout errors, with the `START` room showing the largest number of timeout errors. This behavior indicates that the drone left the `START` room and traveled (at minimum) to the first room in the sequence and then backtracked, as staying in one room would first induce the room timeout error and cause a reset. Additionally, we see Rooms 11 and 12 (c.f., Figures 9, 10, respectively) also inducing global timeout errors. Each of these rooms feature hallways that the drone can easily get "stuck" in without backtracking, leading to the timeout errors. Additionally, the thin horizontal rectangular prism can move laterally as indicated by the double arrow overlay.

Each of these failures indicates that the controller for the drone is quite brittle and struggles with complex situations, where our approach was to be reactive to sensor values and actuate rotors accordingly. As part of our future work we plan to introduce additional controller strategies to

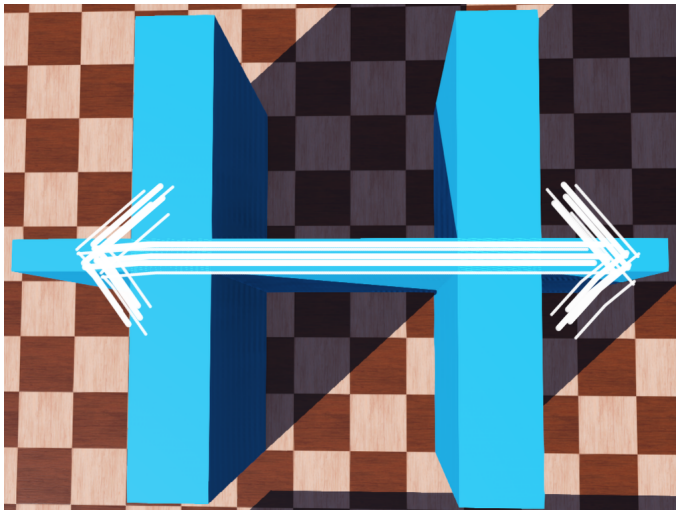


Figure 9: Screenshot of Room ID 11. The horizontal double-arrow indicates lateral movement is possible for the thin rectangular prism.

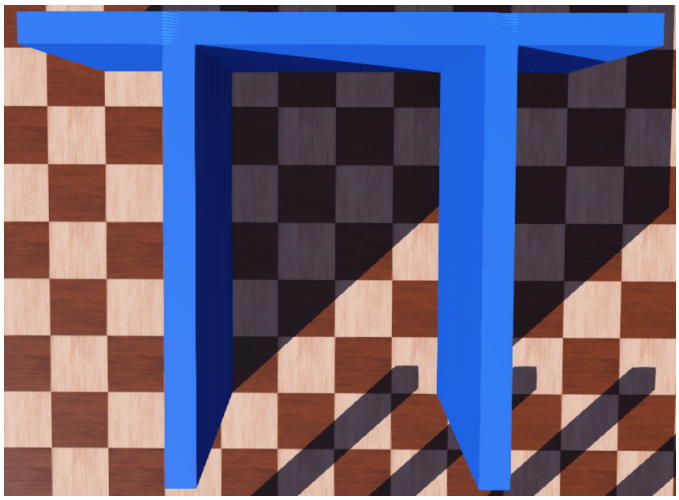


Figure 10: Screenshot of Room ID 12.

better manage the drone control, including using a small-scale neural network to perform base control activities, continuous SLAM mapping to provide information regarding the localized environment [27], and self-adaptation to change path plans and/or controllers when necessary [28].

Lastly, we visualize the paths that the drone took during execution. Figure 11 presents a 3D scatter plot of the (x, y, z) position values recorded over multiple drone runs. The green paths show a `SUCCESS` run, whereas all other runs ended in failure. Notably, the `SUCCESS` runs typically demonstrate a sinusoidal pattern, indicating obstacle avoidance as the drone moves through the maze. `Success (1)` shows significantly less movement along the X axis. Reviewing the logs, this particular run had an optimal room configuration involving only three rooms to navigate, whereas `Success (2)` had to navigate six rooms. The `FAIL` state runs (i.e., `Collision (short)`, `Collision (long)`, `Global Timeout`, `Room Timeout`) all

failed early in the run sequence, denoted by a lack of progress along the X axis. These results again indicate that a more resilient and intelligent controller strategy is needed in future iterations of the project.

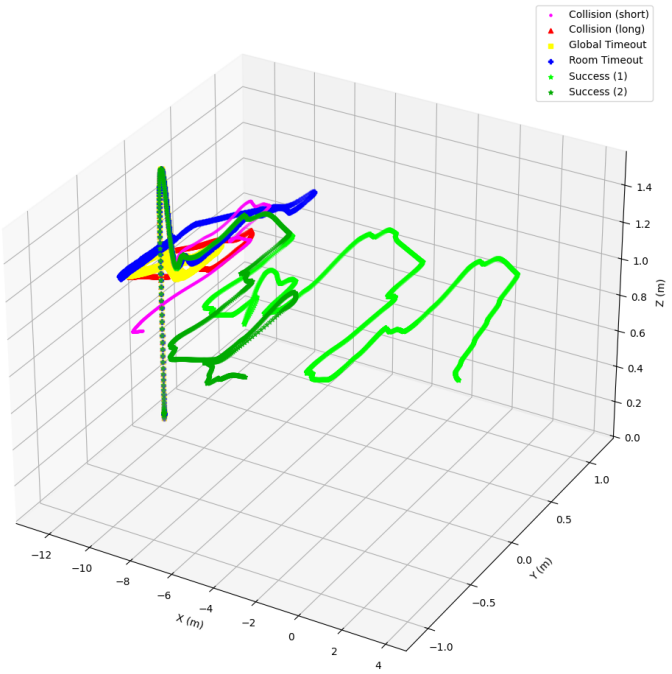


Figure 11: Scatter plot of drone position over multiple runs. The green line is a successful run and the other lines had varying failures.

Figure 12 shows a scatter plot of all drone runs that ended in a `SUCCESS` state. This figure shows a number of drones exploring the progression of rooms with a significant amount of lateral movement. Conversely, Figure 13 shows a scatter plot of all `Course Timeout` and `Room Timeout` states for drone runs. This figure is interesting in that while drones do make limited progress along the X axis (note the truncated X axis values within the figure), they tend to either get stuck or make negative progress. This result suggests that the controller attempted to "back out" to find a better path and was never able to recover. This result suggests that a longer simulation time (as well as a more intelligent controller) may result in additional success states. For presentation purposes we do not include the `Collision` visualizations as the paths essentially fill the available space in the plot.

4.3.3 Threats to Validity

This project was intended as a proof of concept to demonstrate the feasibility of antagonistic development and as such is not a complete study of this development practice in action. As such we have identified the following threats to validity. First, this study focused on a single development strategy between two teams that each comprised an individual graduate student. Such a study is by nature very limited in scope and may not translate well to larger teams in industry. However, in a well-constrained environment with mutual agreement between individuals this process led to a smooth and accelerated development process. Second, the Webots simulation

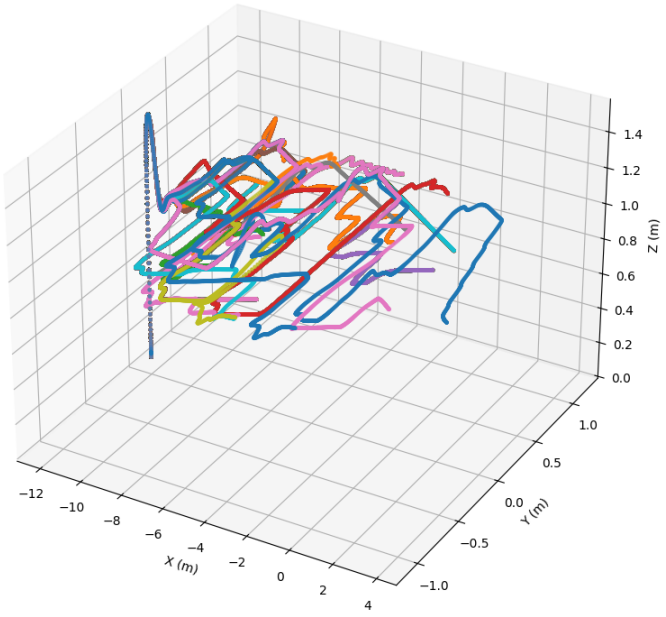


Figure 12: Scatter plot of drone position over multiple runs ending in a `SUCCESS` state. Note that the X axis differs from Figures 11 and 12

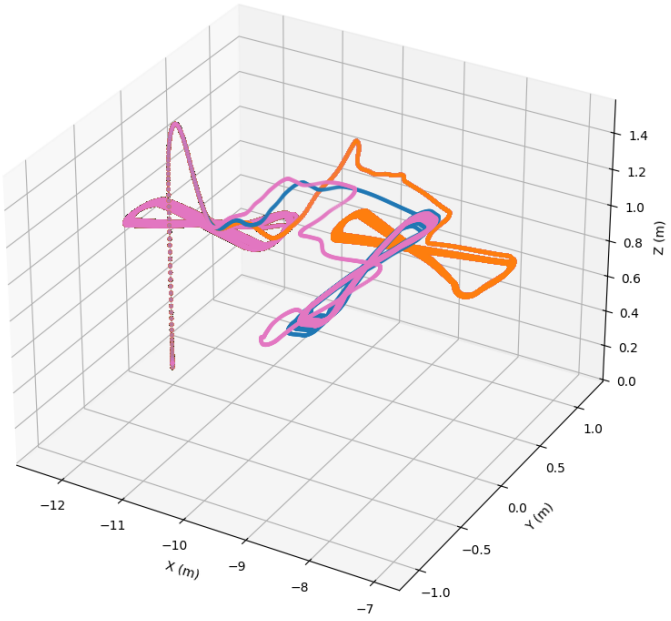


Figure 13: Scatter plot of drone position over multiple runs ending in a `Course Timeout` or `Room Timeout` state.

environment is a self-contained application that does not overcome the reality gap and directly translate to real systems. From our observations, the Webots environment does not realistically simulate physics (e.g., a drone would simply "bounce" off a wall whereas in reality that same drone would catastrophically crash as the rotors hit the wall and forcefully stop). Moreover, the Webots controller code is not directly implementable in the physical robot and must be translated.

Our goal was to provide a proof-of-concept and future studies would benefit from parallel development in both the physical drone and ROS-based simulators (i.e., Gazebo). Third, our configuration of the GitHub repository was minimal and leveraged the automated notification features for all pushes. As such, detailed metrics (e.g., progress on releases, open/closed issues, continuous integration tasks, etc.) could yield a more rigorous experimental evaluation and we leave that for future work as well.

5 Related Work

The antagonistic churn presented in this paper takes an alternative view on contention within the context of a software development lifecycle process. A survey of multiple Agile projects have shown an increase in interpersonal conflict [29]. This conflict is viewed as a negative impact of the Agile development process which can result in reduced productivity [29]. While interpersonal conflict is a possible liability, this paper explores the idea of purposefully antagonistic development within a team of developers to improve development performance. A comparison of related efforts and methods is included in the subsections below.

5.1 Parallel and Competitive Development

Parallel development and prototyping of products has been shown to have positive effects in design and self-efficacy [9]. Similarly, competitive prototyping or even product development has been used to foster innovation in large companies like Amazon [10]. The antagonistic development introduced in this paper differs in scale. While original efforts in parallel development have been viewed within as competition between organizations, or in the case of Amazon, the much smaller "two-pizza" teams [11] that can fed with no more than two pizzas, the competition in antagonistic development is intra-team rather than inter-team. The competition is within a single small team rather than between multiple teams. That is, the competing units are individuals and co-workers rather than teams, groups, or organizations. While competitive prototyping is a well studied topic, the existing literature does not extend to individuals within a single team working on the same product [30, 31].

Within a single project, there has been significant existing work on identifying churn within a software repository [1] focused on identifying improvements that can be made to the software. Issues of defect rates [2], software vulnerabilities [3], maintenance issues [4], architectural violations [5], and general quality issues [6] can significantly impact projects. The introduction of antagonistic development in this work focuses on potential benefits of churn within a single project, rather than negative impacts.

Similarly, **competitive development** is an approach used for the development of projects where multiple teams share a single set of input requirements or other artifacts. This term is more often used in academic settings to describe how teams within a classroom "compete" to deliver the best product for a term project or capstone, typically within the realm of an Agile process. Çulha presented a case study of applying competitive development to an Agile-focused software engineering course, where the intent was to model a competitive learning approach often found in robotics course projects (i.e., which robot most optimally completes a task) [32]. Project success was measured by test cases and team sizes were limited. Watkins and Barnes presented a

comparable approach for including competitive development in academic capstone classes, but focus on describing the Agile methodology in which it was implemented [33]. In this case, project success was measured by an independent panel of faculty to judge the outcomes. While both papers are in the realm of academic software engineering and focus on competition, our approach of antagonistic development aims to provide separate teams with distinct sets of individual requirements/objectives.

5.2 Software Co-Evolution

Co-evolution in the software domain directly parallels our work in that an update to one module may necessitate an update in other modules within the system. Arabat and Sayagh examined co-evolution from the perspective of a large open-source cloud framework (i.e., OpenStack) to determine the impact of updates from one module to another, discovering results both positive and wasteful [34]. Our efforts here aim to be positive in nature, however we acknowledge that without proper management and oversight changes can become wasteful and intentionally vindictive. Zaidman *et al*. focus on the updates to testing activities necessitated by co-evolution, developing views of the interactions between that can be used to support project management and development [35]. Lungu and Lanza present two interactive visualizations of software dependencies between modules as code changes necessitate evolution throughout their respective modules [36]. While we used an open source revision control system with common project management visualization features, our integration and use was limited and could benefit as part of future work to demonstrate change/responsibilities over time. Hebig *et al*. provided both a survey and taxonomy of co-evolution techniques with respect to software meta-models/models [12], focusing on the classification and requirements of each approach. Khelladi *et al*. provided a view of code co-evolution with models and developed Eclipse plugins to support project teams [37]. Our approach focused mainly on source code evolution driven by requirements/objectives. In the future, abstracting our controllers to be model-based can be helpful for minimizing reactionary code changes as a result of team updates. Though similar, antagonistic development differs in that software changes are *intended* to induce an update in other modules to improve functionality, whereas co-evolution does not necessarily specify that updates are required.

6 Discussion

This paper presented antagonistic development as a strategy for developing software with unclear requirements and independent key objectives. Antagonistic development was motivated using a Masters project involving two teams that were required to create an autonomous drone controller and procedurally-constructed environments for validation of the controller. Teams used GitHub as a source control repository and were automatically notified of any changes to the codebase, necessitating internal updates to their respective modules. Significant progress was made over the duration of the 15-week project, leading to a stable proof of concept deliverable from both teams.

Future work for antagonistic development includes broader experiments involving larger teams on more rigorously-defined, industrial-strength projects (i.e., including formal requirements, tests,

etc.), leveraging project management metrics and strategies, direct comparisons of outcomes to other development strategies (e.g., Agile), and a deeper overall analysis of its potential. Additionally, we aim to include project management metrics Future work for our simulation environment includes development of a refined drone controller that can better respond to uncertainty (e.g., neural network-controlled), a framework for automated testing of controllers and procedurally-generated environments in the Webots framework, inclusion of moving/intelligent entities within the simulation that can cause further uncertainty, and translation of our current framework from Webots to ROS/Gazebo for better simulation fidelity.

Acknowledgments

The authors gratefully acknowledge the support of Grand Valley State University.

References

[1] D. Barros, F. Horita, I. Wiese, and K. Silva, "A mining software repository extended cookbook: Lessons learned from a literature review," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pp. 1–10, 2021.

[2] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 24–31, IEEE, 1998.

[3] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[4] G. A. Hall and J. C. Munson, "Software evolution: code delta and code churn," *Journal of Systems and Software*, vol. 54, no. 2, pp. 111–118, 2000.

[5] T. Olsson, M. Ericsson, and A. Wingkvist, "The relationship of code churn and architectural violations in the open source software jabref," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pp. 152–158, 2017.

[6] A. Meneely and O. Williams, "Interactive churn metrics: socio-technical variants of code churn," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–6, 2012.

[7] A. Rahman, J. Stallings, and L. Williams, "Defect prediction metrics for infrastructure as code scripts in devops," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 414–415, 2018.

[8] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[9] S. P. Dow, A. Glassco, J. Kass, M. Schwarz, D. L. Schwartz, and S. R. Klemmer, "Parallel prototyping leads to better design results, more divergence, and increased self-efficacy," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 17, no. 4, pp. 1–24, 2010.

[10] I. Georgousis, "Goliath vs. goliath, it product management at amazon and ibm," in *Forum for Economic and Financial Studies*, vol. 2, pp. 437–437, 2024.

[11] B. Galetti, J. Golden, and S. Brozovich, "Inside day 1: How amazon uses agile team structures and adaptive practices to innovate on behalf of customers," *People & Strategy*, vol. 42, no. 2, pp. 36–41, 2019.

[12] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, 2016.

[13] W. S. Humphrey, "Characterizing the software process: a maturity framework," *IEEE software*, vol. 5, no. 2, pp. 73–79, 1988.

[14] W. S. Humphrey, *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[15] A. Manifesto, "Manifesto for agile software development," 2001.

[16] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338, 1987.

[17] N. Forsgren and J. Humble, "The role of continuous delivery in it and organizational performance," *Forsgren, N., J. Humble (2016)." The Role of Continuous Delivery in IT and Organizational Performance." In the Proceedings of the Western Decision Sciences Institute (WDSI)*, 2016.

[18] M. Hassanalian and A. Abdelkefi, "Classifications, applications, and design challenges of drones: A review," *Progress in Aerospace Sciences*, vol. 91, pp. 99–131, 2017.

[19] Bitcraze, "Crazyflie 2.1," 2024. Accessed July 17, 2024.

[20] O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[21] H. Yang, Y. Lee, S.-Y. Jeon, and D. Lee, "Multi-rotor drone tutorial: systems, mechanics, control and state estimation," *Intelligent Service Robotics*, vol. 10, no. 2, pp. 79–93, 2017.

[22] V. Kangunde, R. S. Jamisola Jr, and E. K. Theophilus, "A review on drones controlled in real-time," *International journal of dynamics and control*, vol. 9, no. 4, pp. 1832–1846, 2021.

[23] S. Mason, C. Stagg, and N. Wardrip-Fruin, "Lume: a system for procedural story generation," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pp. 1–9, 2019.

[24] A. Patel, "Making maps with noise functions."

[25] K. Perlin, "Noise hardware. in real-time shading," *SIGGRAPH Course Notes*, 2001.

[26] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pp. 1–4, 2010.

[27] I. A. Kazerouni, L. Fitzgerald, G. Dooly, and D. Toal, "A survey of state-of-the-art on visual slam," *Expert Systems with Applications*, vol. 205, p. 117734, 2022.

[28] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus, "Architecture-driven self-adaptation and self-management in robotics systems," in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 142–151, IEEE, 2009.

[29] L. Gren, "The links between agile practices, interpersonal conflict, and perceived productivity," in *Proceedings of the 21st international conference on evaluation and assessment in software engineering*, pp. 292–297, 2017.

[30] E. J. Christie, D. D. Jensen, R. T. Buckley, D. A. Menefee, K. K. Ziegler, K. L. Wood, and R. H. Crawford, "Prototyping strategies: literature review and identification of critical variables," in *2012 ASEE Annual Conference & Exposition*, pp. 25–1091, 2012.

[31] B. Camburn, V. Viswanathan, J. Linsey, D. Anderson, D. Jensen, R. Crawford, K. Otto, and K. Wood, "Design prototyping methods: state of the art in strategies, techniques, and guidelines," *Design Science*, vol. 3, p. e13, 2017.

[32] D. Çulha, "Applying competition-based learning to agile software engineering," *Computer applications in engineering education*, vol. 24, no. 3, pp. 382–387, 2016.

[33] K. Z. Watkins and T. Barnes, "Competitive and agile software engineering education," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pp. 111–114, IEEE, 2010.

[34] A. Arabat and M. Sayagh, "An empirical study on cross-component dependent changes: A case study on the components of openstack," *Empirical Software Engineering*, vol. 29, no. 5, p. 109, 2024.

[35] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st international conference on software testing, verification, and validation*, pp. 220–229, IEEE, 2008.

[36] M. Lungu and M. Lanza, "Exploring inter-module relationships in evolving software systems," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pp. 91–102, IEEE, 2007.

[37] D. E. Khelladi, B. Combemale, M. Acher, and O. Barais, "On the power of abstraction: a model-driven co-evolution approach of software code," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering: new ideas and emerging results*, pp. 85–88, 2020.