

“Wanna See Me Do it Again?”

Dynamic Catch Probability Visualization through ManimGL

Ethan Cappelleri, Sam Cowan, Evan Kurtz, Tejas Rama

Abstract

We introduce a new type of replay for highlight and lowlight plays in the outfield by integrating a dynamic catch probability model into an animation of real MiLB plays. This work discusses the case for viewing catch probability over both space and time, and provides example animations as they would be seen on an in-stadium scoreboard. These animations seek to improve fan understanding of how each moment between a hit and catch influences the fielder's probability to make the play. To achieve this, we estimated catch probability at 20 FPS using an XGBoost model, constructed catch probability “clouds”, and implemented 3D player and ball movement by developing a custom, baseball-centric extension to the open-source ManimGL library.

Abstract	2
1 Introduction	4
2 Data	7
3 Training a Dynamic Catch Probability Model	8
4 Animation	10
5 Discussion	14
6 Acknowledgements	14
7 Additional Media Examples	14
Scoreboard Visuals	14
Raw Manim Visuals	14
8 Citations	15
Appendix A: Future work	17
Visualization Improvements	17
Interactivity Improvements	17
Appendix B: Data Limitations	18
Noisy Data	18
Limits of anonymization	18
Appendix C: Modeling Decisions and Evaluation	19
Logistic regression vs XGBoost	19
Why not a time-series model?	21
Model Training and Testing	21
Behavior of the Final Model	22
XGBoost results	23
Appendix D: Feature Engineering	24
Derived Features	24
Wall Approximation	25
Fielder Interactions (in-depth)	26
Direction Handling	26
Appendix E: Animation Technical Details	28
Animation Timeline Creation:	28
Object Movement Through Updaters:	29
Dynamic camera movements with Euler Angle Interpolation:	30
Successes and Failures in Cloud Shape Generation	31
Constructing The Field's Walls and Warning Track	31

1 Introduction

As a baseball fan, when an outfielder makes a spectacular catch or a spectacular blunder, you lock your gaze to the scoreboard and wait for the replay, because you just have to see it again. You can always rewatch Derek Hill getting a good-looking jump and charging in after a ball (see Figure 1), but sometimes the replay lends itself to more questions than it can answer; just how much of an impact did that jump have versus his elite closing speed? When was it that he closed the gap and his chances shifted from extremely unlikely to a near certainty?



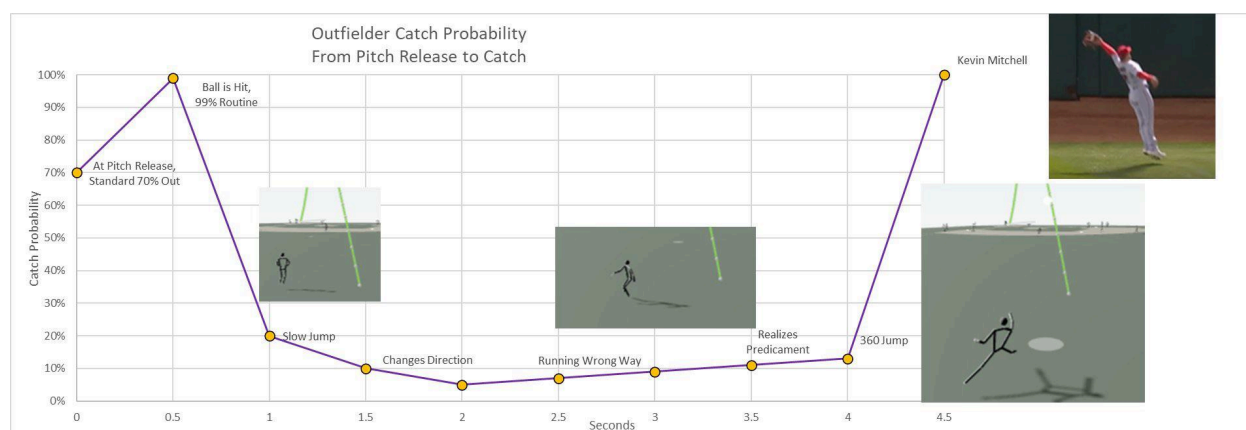
Figure 1: Derek Hill makes a diving catch in the outfield, June 29, 2024 (video linked)

Statcast gives this play a 40% catch probability¹, which means that given the conditions at the start of the play, the average fielder would catch this twice in every five attempts. However, a spectator sees and cares about more than just the moment the ball is pitched². A player stumbling or accelerating clearly changes a catch's likelihood mid-play. At the same time, a player shifting slightly to his left or right could drastically affect catch probability in a way you wouldn't be able to see with the naked eye. With this in mind, we set out to model and visualize catch probability as a dynamically updating value over time and space.

¹ <https://baseballsavant.mlb.com/savant-player/derek-hill-656537?stats=statcast-r-fielding-mlb>

² Statcast predicts catch probability at the moment a pitch is released instead of the moment it is hit, to account for the fact that some fielders know better than others which side of the field a pitch will be hit to, and thus gain a step in that split-second before the pitch

3



4

Figure 2: One of Statcast's public displays of dynamic catch probability

That's not to say Statcast hasn't considered dynamic catch probability – they have – Figure 2 shows a Tom Tango-produced proof of concept, taking a snapshot of catch probability every half-second over a play, and allowing that to tell a story. Unfortunately, this type of chart only appears in the occasional Twitter post.

This work builds on Tango's storytelling approach, generating appealing and accessible visualizations for a variety of plays and the second-by-second changes in probability that make them exciting.

Through our animation, fans can watch catch probability change throughout a play in a way that is intuitive yet analytically rigorous. We achieve this by reimagining catch probability as a cloud centered around the ball's final landing point, covering the parts of the field which the player could range from in time to make the catch. As the play moves forward in time, the cloud shrinks (see Figure 3). With this perspective, outfield plays can be followed as a player trying to chase down the ball's cloud. The better he can keep himself within the cloud, the more likely he is to catch it.

³ https://www.reddit.com/r/baseball/comments/127lk7m/tom_tangos_catch_probability_chart_for_that/

⁴ <https://x.com/tangotiger/status/1717544141712699779Figure>

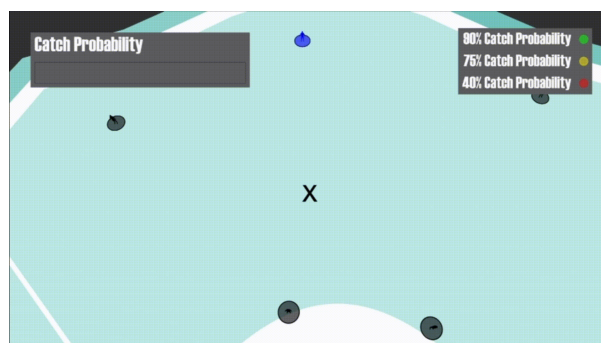


Figure 3: (Video) The center fielder's shrinking catch probability cloud throughout the play

Powering this animation is real-time player-tracking data and a dynamic catch probability model. While we don't have tracking data for that exact Derek Hill play, Figure 4 follows a similar play as a proxy for what a fan could see after that catch:

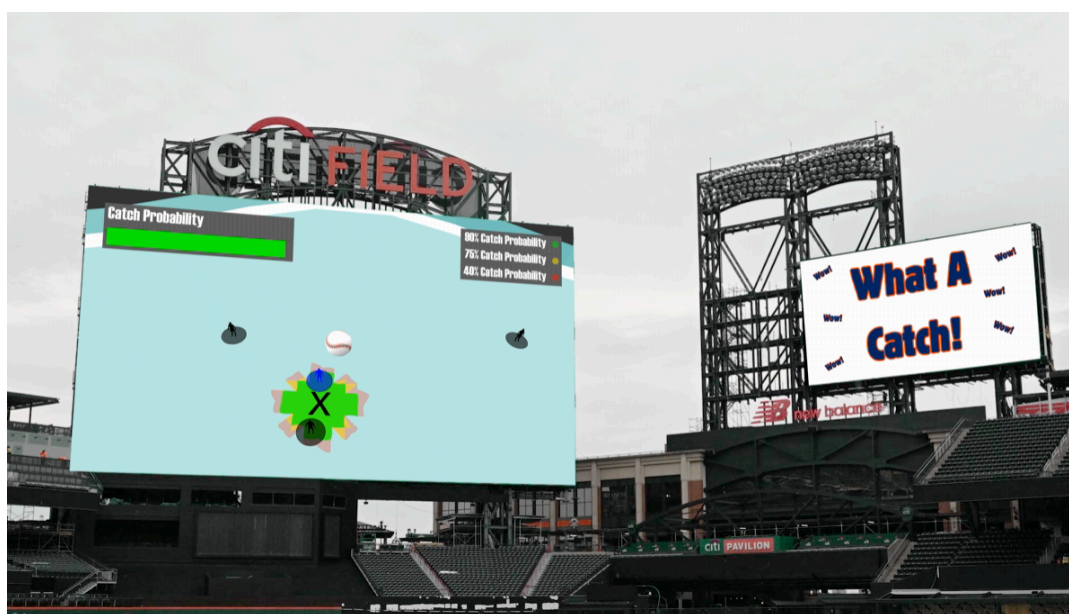


Figure 4: (Video) The scoreboard shows proxy Derek Hill making a difficult charging catch

Proxy Hill's jump is good enough to start him off in the red part of the cloud – he's just above a 40% chance of making the catch but still has work to do. He covers 121 feet over seven seconds, and in the last second consecutively crosses the 75% and 90% thresholds (orange and green). Even as the cloud of viable catches shrinks, he keeps the pace necessary to stay inside it, calls off his shortstop, and makes a running catch. As the fielder catches his breath, fans notice that his jump was about average, but his last-second closeout was immaculate.

Highlights and lowlights alike can be animated in this manner and provide for engaging videos on the scoreboard, as seen in Figure 5.



Figure 5a: (Video) LF is within the 90% green zone (video linked)

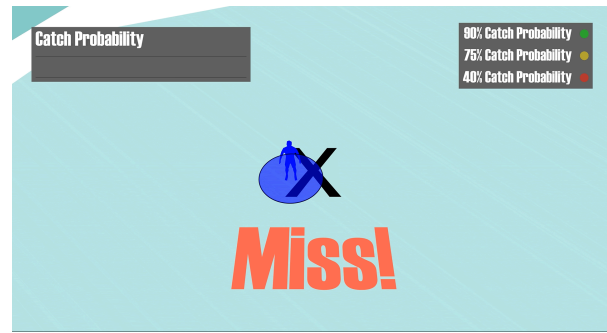


Figure 5b: Fans see in context how the LF overplayed the ball's flight

2 Data

SportsMEDIA Technology (SMT) provided us with anonymized player and ball-tracking data from minor-league games at four stadiums over the course of one unidentified season in the last decade⁵. Within this data, we found over 1,300 plays in the outfield. These were used for training our model and animating sample plays (Figure 6).

Play Distribution		
Counts of plays by type and fielder		
	Count	% of Total
Plays	1354	
Catch Totals		
Catches	840	62.0%
Misses	514	38.0%
Plays by Fielder		
LF	409	30.2%
CF	480	35.5%
RF	465	34.3%
misc		
Total Wall Plays	130	9.6%

Figure 6: Summary statistics about our filtered dataset

The data contained markers for timestamps when the ball was hit into play and when it was acquired by a fielder, which we took as the start and end time of the play for our purposes.

⁵ Data is recorded at 20 frames per second

Catches were defined as plays where an outfielder acquired the ball before it took a bounce (off the ground or wall), and we filtered our data to only timestamps from the moment of contact to the ball hitting the glove or ground. We studied plays with a launch angle between 15 and 50 degrees.⁶

3 Training a Dynamic Catch Probability Model

To build a meaningful probability cloud, we trained a real-time catch probability model, which follows the pursuing outfielder most likely to make each play. At each data point within a play, we predicted the chance that the fielder would catch the ball at the end. We extracted 10 features⁷ calculated from ball and player kinematic information, and the ball's final landing spot (see Figure 9). Notably, we controlled for the impact of proximity to an outfield wall, which can make a catch much more difficult than a player's distance to it would otherwise suggest. For a complete explanation of our model features, see Appendix C.

By the end of a play, the predicted probability often converges to 100% catch or no catch (Figure 7), apart from close finishes like when a player makes a diving grab.

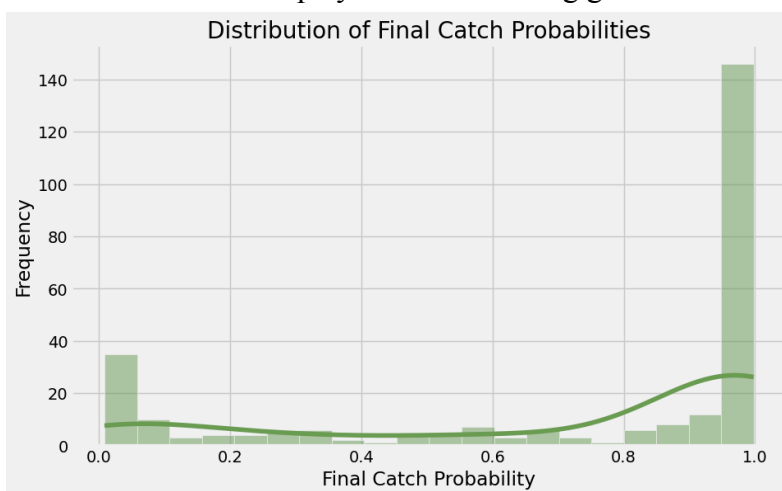


Figure 7: Frequency of predicted catch probabilities on the final timestamp of plays in our data. Most plays converge to catch or no catch.

⁶ per Statcast, balls between 10 and 25 degrees launch angle are classified as a line drive, and 25 to 50 as a fly ball. We started at 15 to exclude low line drives that might be catchable by infielders.
<https://www.mlb.com/glossary/statcast/launch-angle>

⁷ The ten features in brief: Remaining hangtime, remaining distance to the ball, player speed, wall distance at end of play, angle to the ball in cosine and sine, projected distance run (given current speed), required speed to reach ball, difference in ball path from projected trajectory

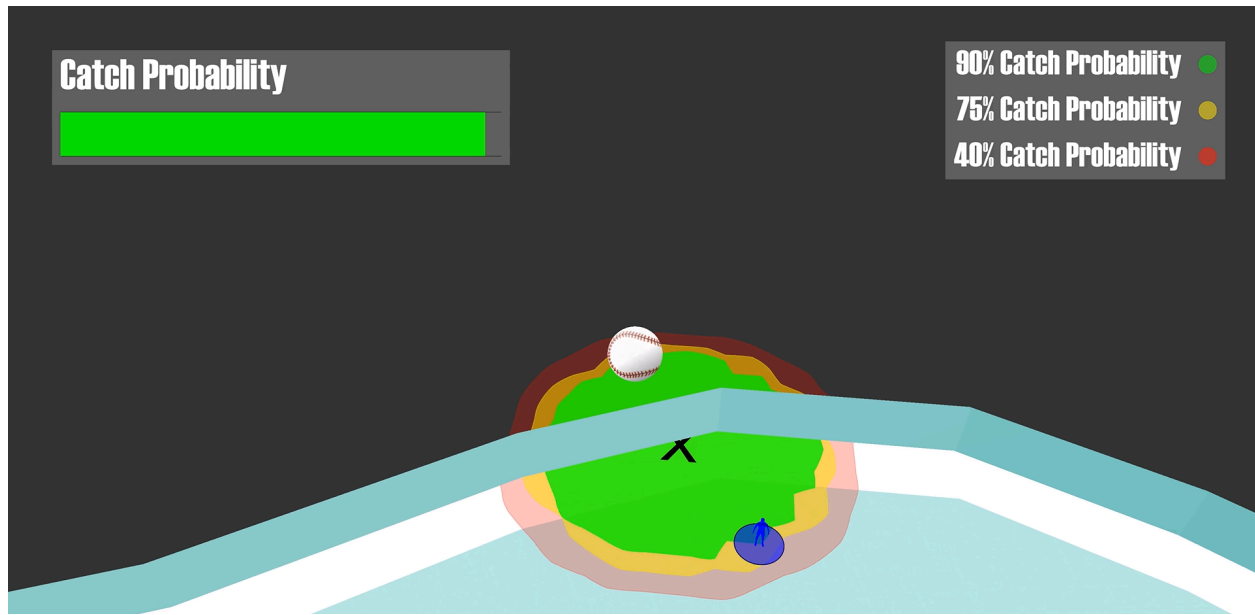


Figure 8: Controlling for the proximity of the wall keeps us from overestimating the catch probability on wall plays.⁸

Sample of All Timestamps Data									
10 rows of engineered features for an example play									
remaining_hangtime	remaining_distance	player_speed	wall_distance	cos	sin	projected_distance	required_speed	distance_will_cover	trajectory_difference
3750	169.89	0.71	0.72	-0.93	-0.38	167.24	45.3	2.65	13.01
3700	169.87	0.7	0.72	-0.93	-0.38	167.27	45.91	2.6	13.15
3650	169.85	0.69	0.72	-0.93	-0.38	167.33	46.54	2.53	11.7
3600	169.84	0.68	0.72	-0.93	-0.38	167.4	47.18	2.43	11.49
3550	169.82	0.65	0.72	-0.93	-0.38	167.53	47.84	2.29	12.3
3500	169.8	0.62	0.72	-0.93	-0.37	167.65	48.52	2.15	10.1
3450	169.79	0.59	0.72	-0.93	-0.37	167.75	49.21	2.04	10.15
3400	169.77	0.58	0.72	-0.93	-0.37	167.82	49.93	1.96	10.55
3350	169.76	0.56	0.72	-0.93	-0.37	167.88	50.67	1.88	9.36
3300	169.75	0.56	0.72	-0.93	-0.37	167.9	51.44	1.84	9.32

Figure 9: Table representation of the features we used to predict catch probability

We wanted a model as sensitive as possible to what the player is doing in the exact moment of prediction, so that fans can easily pinpoint inflection points in catch probability. As a result, we chose a stateless model,⁹ specifically an XGBoost classifier. While logistic regression often leads to a more intuitive binary classifier, we found that training it to predict catch probability did not produce intuitive feature weights (Appendix B1)

⁸ The cloud extends beyond the walls because the model isn't aware of where the walls are, just that they are nearby

⁹ See Appendix C.4 (noting in this paper it says C but we have 2 Cs, this links to C.4 in your TOC)

Like other tree-based algorithms, XGBoost creates highly non-linear and jagged decision bounds (Figure 10). Crossing these thresholds can lead to unpredictable and sharp changes in predictions.

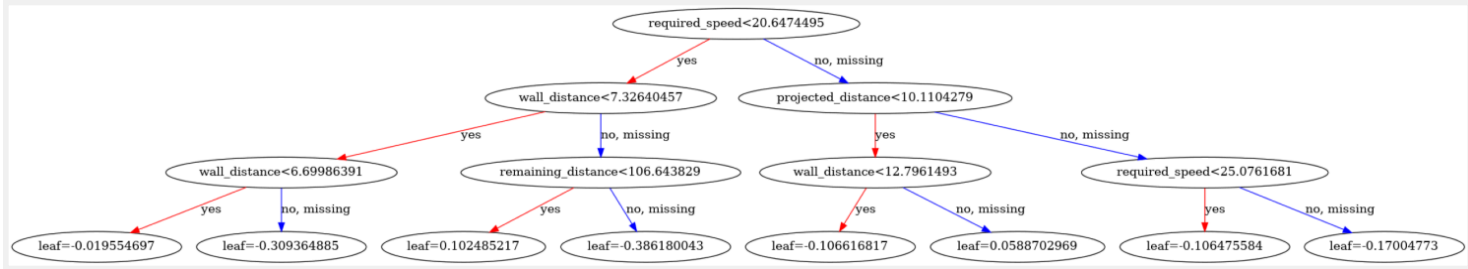


Figure 10: One of the 500 decision trees that comprise each of our XGBoost catch probability models

To better align predictions with intuition, we imposed monotonic constraints while training. Holding other variables constant, increasing remaining hang time is constrained to increase catch probability, while increasing remaining distance is constrained to decrease it. These constraints helped prevent unrealistic behavior and produce intuitive probability surfaces for visualization. These are basic rules of how catch probability is perceived, even if there are a few edge cases where breaking them might improve accuracy.

4 Animation

Visualizing the complex intersections of these features requires a comprehensive animation system capable of handling a variety of situations and rapidly changing values while still producing a meaningful and accessible visual. We used the ManimGL¹⁰ library, an open-source math visualization system built by Grant Sanderson of 3Blue1Brown, as our rendering engine, and customized it to power 3D baseball visualizations complete with dynamic camera work and analytical overlays. We plot our players based on their real-life points and convert the coordinates relative to the screen size, maintaining locational accuracy while empowering high-quality visualization in half speed (see Figure 11).

¹⁰ There are two versions of Manim. All references to Manim in this paper are about ManimGL. This is the more powerful, but worse documented version of the library made by and for 3Blue1Brown’s Grant Sanderson. There is a community driven and better documented version called ManimCE, but it was not used in the final output of this submission.



Figure 11: An overview of the field construction, specifically the Oakland Coliseum¹¹¹²

From this foundation, we build out the visuals using our model. At each timestamp, we construct a series of “ghost fielders” that inherit the kinematic features of our real player at offset positions, logging the catch probability to represent our fielder’s chances if he were in a different location. This led to an important decision: whether to make the cloud center the player’s live location or the ball’s final location. That is, whether to track the ball moving into the fielder’s range, or the fielder working to place himself in position to make the play.

Centering the ghost fielders around the ball’s landing point provides two advantages in interpretability. First, the average fan focuses on the player himself. Placing the player in the red zone (the border between success and failure) rather than the ball keeps the most meaningful part of the animation in the same visual field as a fan’s focus. Second, the paradigm of the player racing to make up ground and move towards the green cloud provides a desirable layer of tension to the experience. Rather than focusing on whether the ball - an object with relatively constant motion - will slip into the high-probability zone, we place the player, who has agency over their own actions, in the spotlight. This replicates the real-time experience of watching a highlight, enjoying the player’s skill as they convert on a low-probability play.

¹¹ SMT’s NDA doesn’t allow us to look at the dimensions of the stadiums in the dataset, so we picked the Oakland Coliseum for our animation’s stadium.

¹² Stadium perimeter constructed from real-world measurements to accurately visualize the outfield wall and foul territory. For additional details, see Constructing the Field’s Walls and Warning Track in Appendix D.

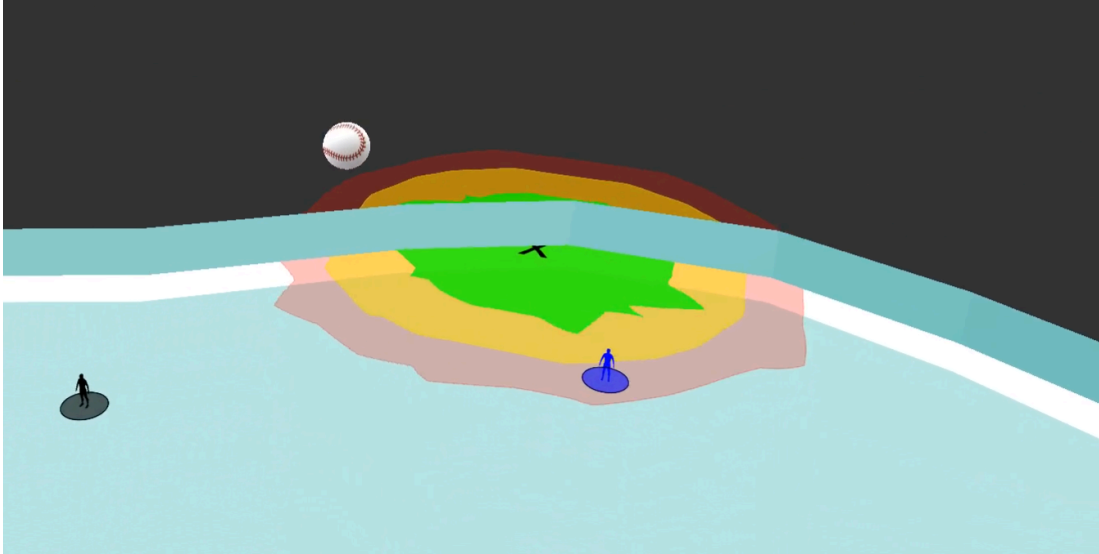


Figure 12: *A player working towards a difficult ball, currently in a low-probability position*

These clouds themselves are also unique, forming noisy concentric circles as the play continues. This was an intentional choice to maintain the representability of the animation relative to real-life play. Players are not perfectly able to move and catch the baseball in all directions and positions, and therefore the clouds should not be perfect either (Figure 12). To construct these visuals, we analyze the set of ghost fielder points at each timestamp as a series of polar points relative to the landing location. For each cloud, only the points with a catch probability greater than the given threshold are selected. With that point set, we spin in a circle, taking 10-degree segments. For each segment, we identify the valid point with the largest radius, combining these points to approximate the total perimeter of the cloud.

Catch probabilities were also smoothed to improve visual coherence, particularly because the cloud shape generation involves assuming the geometry between boundary points (Figure 13). This smoothing process involves reweighting each ghost fielder's catch probability to account for the noisiness of the nonlinear model. Each visualized point is weighted 30% by its own predicted probability and 70% by the average probability for all points of the same radius. This gives space for the model's individual variation in predictions to produce non-circular clouds, while reducing the noise between points that should have roughly equal catch probabilities.

Similar to how monotonic constraints were applied to our model to align results with natural intuition, we applied two rules to the predicted points:

1. If catch probability ever goes below 10% for a given offset, we assume it is 0% for the rest of the play, as it would have to jump up to 40% to ever be colored in again. This prevents extreme moments of noise from the model from causing sudden flashes where one bad datapoint makes a large chunk of the field suddenly appear to be a catchable location.

2. Catch probability at an offset can never be 20% higher than the lowest it has been during a play. This is reasonable, as aside from sudden changes in speed or direction, the catch probability at a fixed distance should only go down as remaining hangtime decreases.

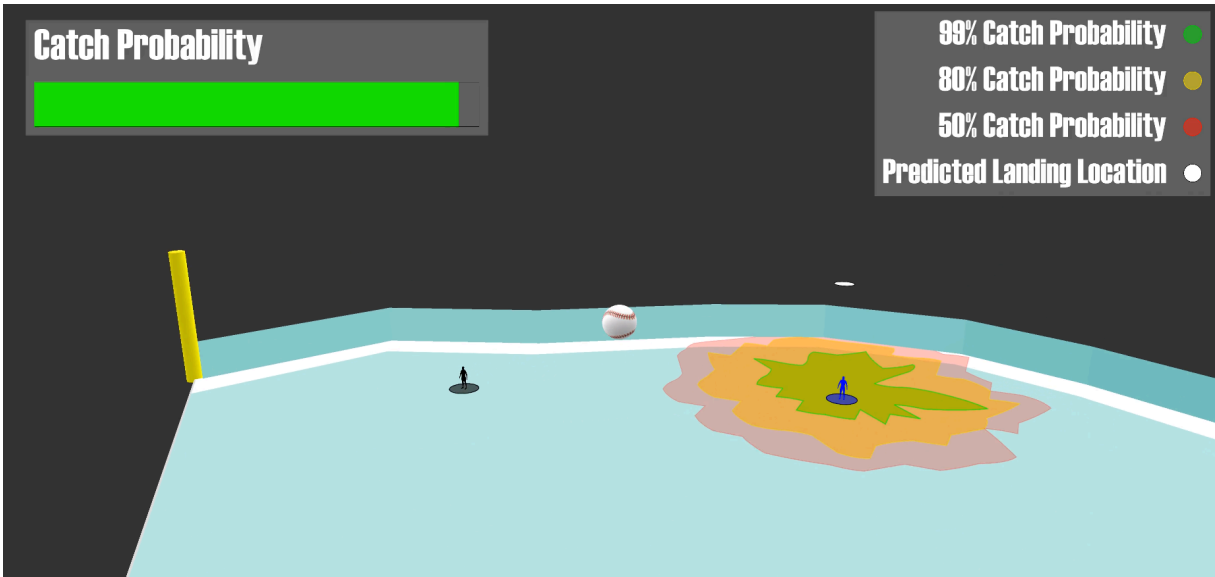


Figure 13a: An older model without smoothing produces sharp corners and confusing clouds

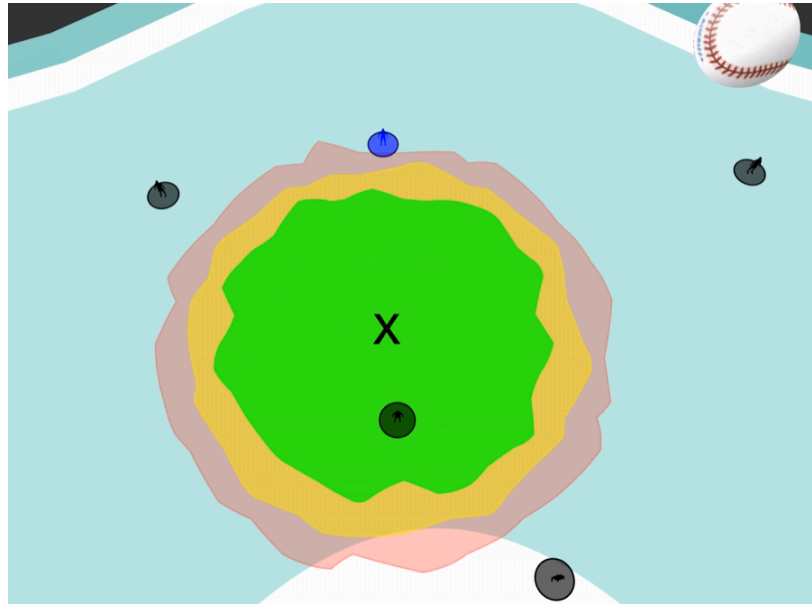


Figure 13b: Smoothing greatly improves visual clarity

These decisions were made with the average fan in mind. By centering the player as the object of action in the animation, using universal colors to represent high and low probability, and smoothing out the rough edges of a complex model (Figure 13a, 13b), our visualization can

be intuitively understood by a variety of fans and remains comprehensible even from a distance or at awkward angles throughout the park.

5 Discussion

Teams are rapidly innovating their in-stadium scoreboard graphics and fan experiences to engage spectators, while those same fans become more familiar with advanced analytics. Since the inception of batting average, these metrics crafted by analysts have leaked into mainstream discussions, coloring our understanding of the modern game. Savant stats like xwOBA and average exit velocity are now even appearing in broadcasts and stadium scoreboards themselves. We believe our animation is a natural next step.

6 Acknowledgements

We would like to thank Dr. Meredith Wills, Andrew Steenkamer, and Billy Fryer for their assistance throughout this project.

Our 3D visuals took inspiration from Zac Rogers' PlayerTV project for the 2022 Big Data Bowl, which visualized NFL kickoff returns in 3D from the perspective of the returner.

The Python code producing the 2D animation in the appendix was created by Yu-Hao Yeh, with minor changes made by the authors

7 Additional Media Examples

Scoreboard Visuals

[Near-Wall Catch](#)

[Charging Centerfielder Catch](#)

[Left Fielder Miss](#)

[Full Screen Visual](#)

Raw Manim Visuals

[Charging Centerfielder Catch](#)

[Left Field Miss](#)

[Cloud Updating Example](#)

8 Citations

3b1b. (Feb 14, 2021). *GitHub - 3B1B/manim: Animation engine for explanatory math videos · GITHUB*. Github. <https://github.com/3b1b/manim>

Andrew Clem. (Sep 1, 2023). *Clem's baseball ~ Oakland Coliseum*. Clem's Baseball. <https://www.andrewclem.com/Baseball/OaklandColiseum.html>

Andrew Fox. (Jan 29, 2015). *Complete Outfield Dimensions | Community Blog*. Fangraphs. <https://community.fangraphs.com/complete-outfield-dimensions/>

Diego Osborn. (2023). *SMT Data Challenge Paper - Sports Analytics*. Github. <https://sportanalytics.com/wp-content/uploads/2025/03/SMT-Data-Challenge-Paper.pdf>

Major League Baseball. (May 26, 2017). *Catch probability updated to include direction*. MLB.com. <https://www.mlb.com/news/catch-probability-updated-to-include-direction-c232532408>

Major League Baseball. (n.d.). *Field dimensions | glossary | mlb.com*. Major League Baseball. <https://www.mlb.com/glossary/rules/field-dimensions>

Major League Baseball. (n.d.). *Launch Angle (LA) | glossary | mlb.com*. mlb.com. <https://www.mlb.com/glossary/statcast/launch-angle>

Manim. (2020). *Surface - Manim community v0.20.1*. Manim Community. https://docs.manim.community/en/stable/reference/manim.mobject.three_d.three_dimensions.Surface.html

Net World Sports. (n.d.). *Baseball Field Dimensions & Layout Guide | NET World Sports*. Net World Sports. <https://www.networldsports.com/buyers-guides/baseball-field-dimensions-guide>

Robbie Dudzinski. (Aug 6, 2025). *Drawing precise MLB field outlines in Python with polar coordinates | by Robbie Dudzinski | Medium*. Medium. <https://medium.com/@robbiedudz34/drawing-precise-mlb-field-outlines-in-python-with-polar-coordinates-8abd1542a34a>

SMTSubmission2026. (26 July, 2026). *Manim Video Library*. YouTube. https://www.youtube.com/playlist?list=PLdS_gDhLhIWs

u/pjokinen. (n.d.). *R/baseball on Reddit: Tom Tango's catch probability chart for that Renfroe Grab Yesterday: "either the best or the worst defense ever depending on when you start the clock."* Reddit.

[https://www.reddit.com/r/baseball/comments/127lk7m/tom_tangos_catch_probability_chart for that/](https://www.reddit.com/r/baseball/comments/127lk7m/tom_tangos_catch_probability_chart_for_that/)

Tango, T. (2021). *Dynamic Catch Probability* [Post]. X,
<https://x.com/tangotiger/status/1717544141712699779>Figure

Appendix A: Future work

Visualization Improvements

Fans would benefit even more from a graphical representation of the catch probability over time, rather than the current bar that visualizes for each frame. We envision replacing the current bar with a graph similar to the recent FIFA “heartbeat” graphic in the World Cup. If possible, there were limited resources available in the Manim community on how to create such a graph overlay with the limitations present in our framework. Without this, we can still view what such a graph would look like at the end of the play through Figure 2



Figure A1: FIFA's heartbeat graph

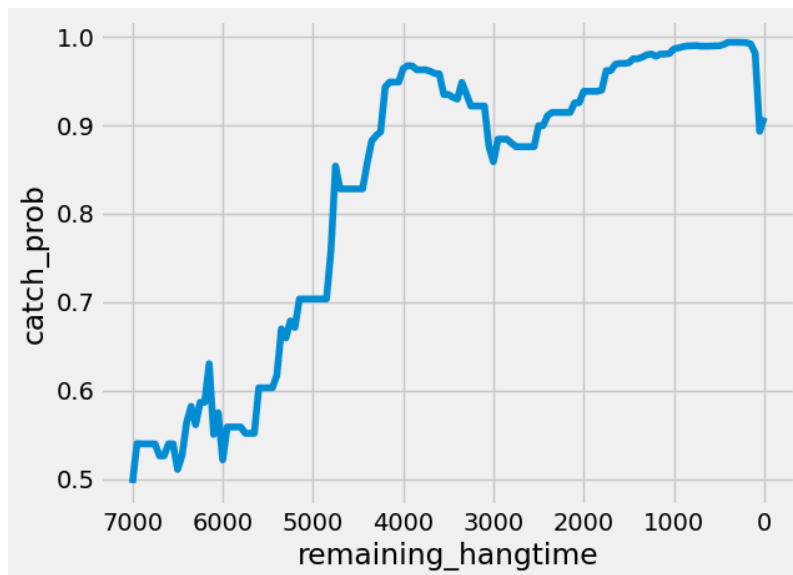


Figure A2: Catch probability moves up and down throughout an exciting play

Access to video replay or more detailed field modeling data could also enable us to overlay our clouds onto real-life footage to better show fielder movement

Interactivity Improvements

Empowering the graphics and fan experience team remains a commitment of this project. While our current Manim system allows for freeze-framing and scrubbing to highlight key moments,

creating tools to visualize theoretical plays in which a fielder reacts differently would further improve the fan experience. We envision an implementation similar to Statcast overlays on broadcasts and scoreboards, and could highlight specific moments where fielders make a good or bad move.

Appendix B: Data Limitations

Noisy Data

The data came with imperfections and noise from the camera-based tracking system that produced it. We found one prominent issue where fielder positions were missing for certain timestamps and replaced with their position at the preceding timestamp, leading to sudden spikes in our speed calculations, which confused our prospective models until we figured out the cause. Other plays existed (as depicted below) where the player and ball timestamps were out of sync. As a result, some of the plays the model thinks are most interesting are just routine plays with bad underlying data.

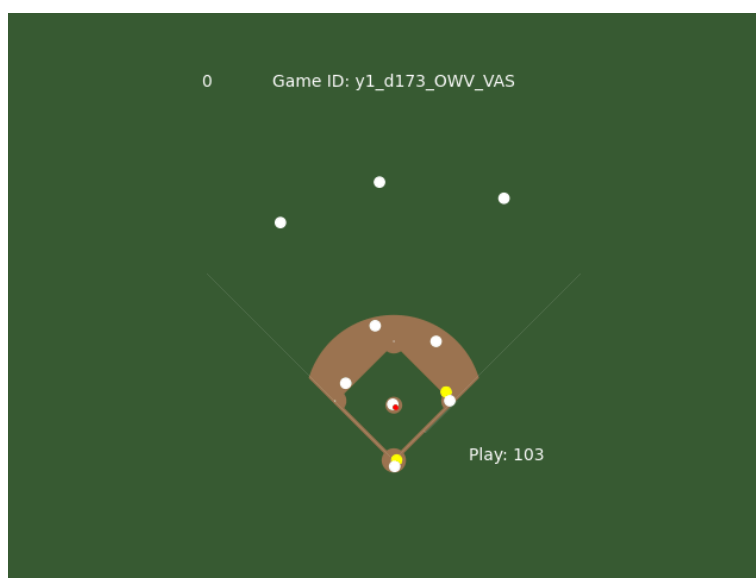


Figure B1: Example play where player data is out of sync with the ball.

Limits of anonymization

Ideally, we would optimize our model around understanding the plays we want to depict: extraordinary catches. However, our data is anonymized, and while we do have tracking data, we

can't watch game footage of these plays to truly understand every basic fact of the play. While we could develop heuristics to search for plays like dives or jumps, we found it difficult to identify the exact behavior systematically. Additionally, the tracking system generating the data seemed to struggle particularly with these plays, as apparent dives would occlude the ball from the camera system for several frames. (Figure B2).

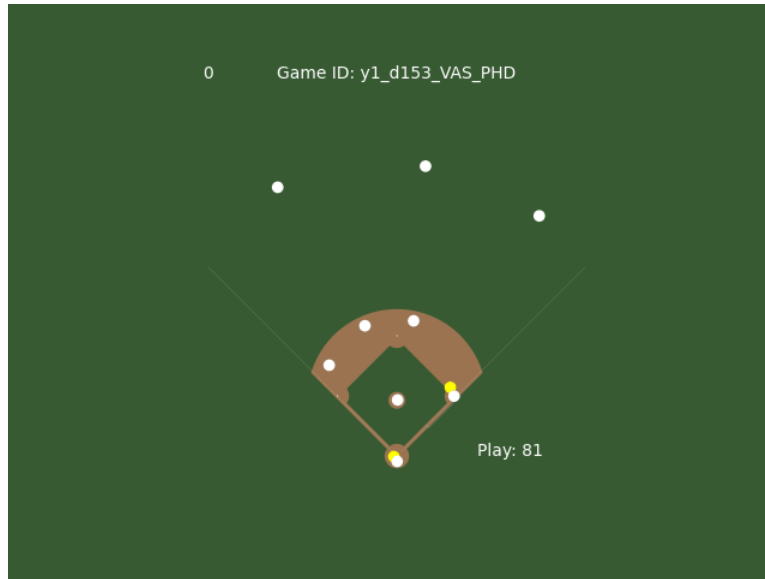


Figure B2: (Video) Suspected fielder dive where the tracking system loses the ball

Appendix C: Modeling Decisions and Evaluation

Logistic regression vs XGBoost

One major decision we had to make was which model best suited the animation. We had competing XGBoost and logistic regression models that scored similarly (XGBoost had the edge in Brier score), but behaved vastly differently. Logistic Regression makes decision predictions through transforming a linear combination of features. It resulted in very uniform sigmoid results when plotted over hangtime (see Figure C1).

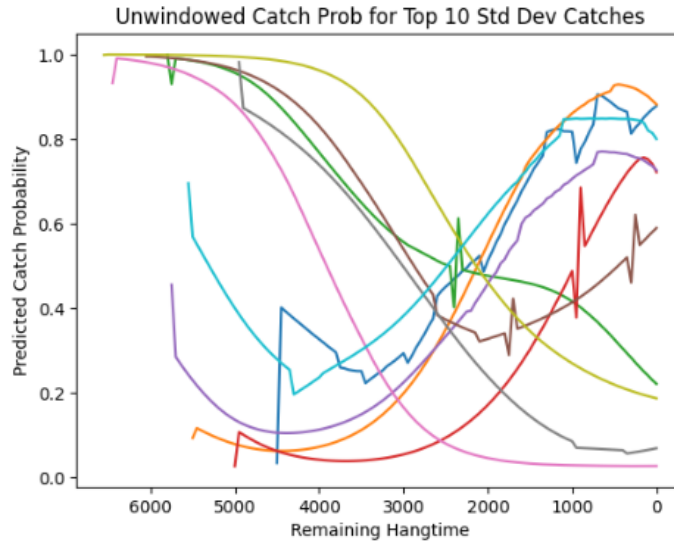


Figure C1: The highest standard deviation plays from an early logistic regression model all follow a sigmoid pattern of catch probability over the play

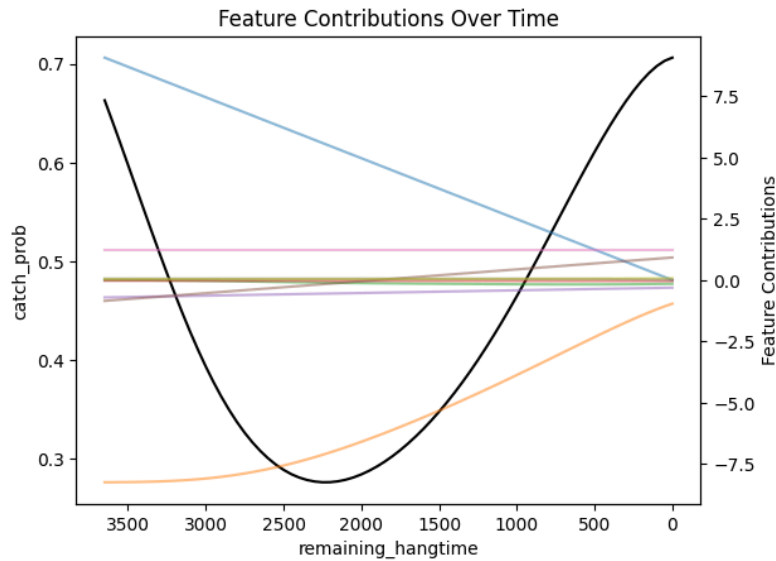


Figure C2: Since we are plotting over remaining hangtime, its contribution as a feature (light blue line) linearly descends throughout the play. On the other hand, the remaining distance has a negative weight (orange), and as the fielder accelerates and closes the distance, we see its contribution rise.

While logistic regression is a simpler model and yields more predictable behavior, its predictions are not very sensitive to player movement. We specifically wanted to highlight bursts of speed, last-second dives, or bad jumps saved by a quick pivot. For this reason, the tree-based XGBoost model was chosen for the final display. That said, revisiting logistic regression should be considered in conjunction with further feature engineering, as it has the higher potential of the

two to produce visually intuitive results. The behavior of our logistic regression models wasn't as dynamic as the XGBoost models, but it was significantly smoother. If we're more intentional with our features, using non-linear interaction terms between features could help bridge the accuracy gap.

Why not a time-series model?

While it may be intuitive to treat plays as a time series, we picked a stateless model (XGBoost) which only has access to a single row for each prediction. This is because we wanted our model to only react to changes as they happen, as opposed to propagating information learned earlier in the play slowly throughout it. With a stateless model, the model has to make a new decision at every timestamp, which forces it to decide if the changes in inputs are enough to warrant a change in outputs. This way, new information (e.g., player changes speed) is never overshadowed by already-known information (e.g., "the catch was 3% based on initial timestamp so now that a second has passed and nothing huge has happened, let's disregard that%")

Time-series models also don't exactly fit our needs because we are counting down to a terminal event (ball hits ground), not predicting arbitrarily into the future as such models are good at.

Model Training and Testing

We capped our trees' `max_depth` at 3 to only allow for 3 degrees of feature interaction, limiting the risk of the model overfitting.

Hyperparameters selected:

- # estimators: 500
- Max depth: 3
- Learning rate: 0.1
- Monotone constraints: Positive relationship with remaining hangtime, negative relationship with remaining distance
- Minimum child weight: 20
- Early stopping rounds: 10

To avoid data leakage from the fact that many plays are part of the same play, we split up full plays between training, test, and validation sets, so that no two snapshots from the same play appear in both training and testing. We first set aside 20% of our data as a test set using SKlearn's `GroupShuffleSplit` function, and trained 5 models on different subsets of the remaining data with sklearn's `GroupKFold`. Our final model is an ensemble averaging these 5.

Behavior of the Final Model

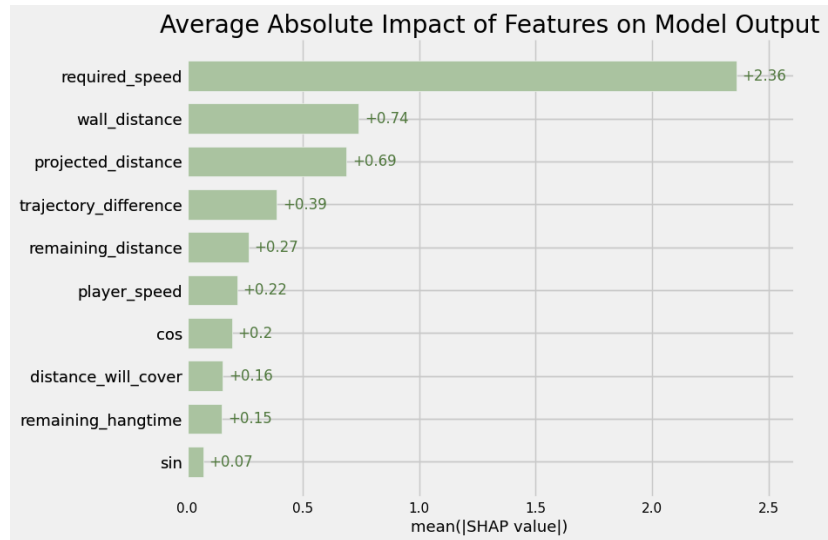


Figure C3: *Shapley values for one of 5 XGBoost models used in the ensemble. required_speed is considered the most valuable feature by all five models. Shapley values are a way of estimating how much each feature contributed to a model.*

XGBoost results

A final evaluation of our model shows it is strong, although considerably better at predicting catches than non-catches due to the class imbalance seen in Figure 10 (62% of plays in our dataset are catches). Additionally, most catches are relatively routine and have high accuracy by default, which leads to 97% of its predicted catches being correct.

Classification Report				
Log Loss: 0.181 Brier Score: 0.045 Accuracy: 0.942				
	Precision	Recall	F1 Score	Support
No Catch	0.936	0.873	0.904	7,509
Catch	0.945	0.973	0.959	16,771

Figure C4: *classification report for XGBoost. Both precision and recall are high, indicating the class imbalance was handled properly*

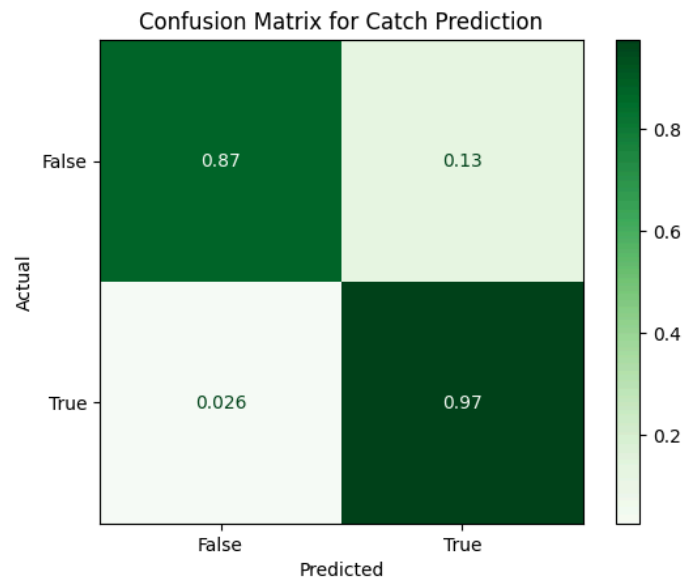


Figure C5: *The model's confusion matrix*

Appendix D: Feature Engineering

Derived Features

Remaining Hangtime

The time in milliseconds left before the ball hits is caught or hits the ground

Distance Remaining

The distance in feet between the player's current position and the ball's final landing position.

Player Speed

The ft/s speed the player has moved at on average in the last 50 milliseconds

Wall Distance

The approximated distance from the player to the nearest wall at the end timestamp of the play. Plays near the wall are more generally more difficult (players don't want to run full speed into the wall), so this is meant to provide a nearly constant reduction in catch probability. We use the final timestamp's wall probability throughout so as to not hold out crucial information (that a catch is against the wall) from the model for most of the duration of the play.

Cos

Cosine of the angle from the player to the ball (where 90 degrees is dead center field)

Sin

Cosine of the angle from the player to the ball (where 90 degrees is dead center field)

Distance will cover

Calculated as the player's speed multiplied by the remaining hangtime. The distance the player would run if they maintained their current speed for the rest of the play.

Required Speed

The minimum speed required for a player at this position to reach the ball's final position.

We note that while Statcast's catch probability model is not made public, their documentation on handling direction states that they increase the feet per second required to make the catch by 1 ft/s for plays behind the fielder, which suggests they use a similar feature. That isn't too surprising because required speed produces the highest Shapley value of all our features.

Required speed can clearly delineate the boundary of when a catch becomes impossible. If a player has to run at 35 ft/s for the rest of the play to make the catch, they have virtually zero

chance of making the catch. It also pairs nicely with the fielder's current speed in the tree-based model, as decisions can be made on whether the fielder's current speed is close enough to the required speed.

Projected Distance:

If the player were to run at their current speed in their current direction, this is their distance from the ball's final landing position. Both being too slow and overrunning the ball would cause high projected distance.

Trajectory Difference

We calculated the expected landing point of the ball throughout the play based on its movement without taking into account wind, spin, or baseball-specific physics such as seam-shifted wake. This assumed gravity was the only force acting on the ball. Trajectory difference is the distance from this projected landing point to the actual landing spot, and converges to 0 at the end of the play. This was used as a proxy for how much adjustment outfielders had to make throughout the tracking of the ball.

For every timestamp after a ball is hit, we take its current height, velocity, and gravitational acceleration (~ 32 ft/s), solve for the time when it will hit the ground, and extrapolate the ball's velocity and acceleration in the X and Y planes to the end of the play. For high pop flies with limited wind, this is almost an exact approximation of the ball's landing point from the start of the play. For line drives with lots of backspin, this will always underestimate the distance the ball travels.

Wall Approximation

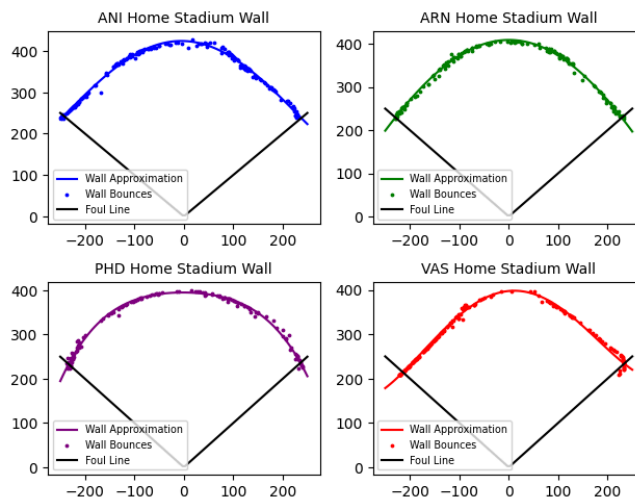


Figure D1: *Quartic approximations of the walls at each stadium in the dataset*

As the data were anonymized, we weren't permitted to know the exact field dimensions of the four stadiums in the dataset when calculating distance from the wall. We built on Diego Osborn's 2025 SMT Data Challenge paper, which uses quadratic approximations of outfield walls to estimate wall distance for a catch probability model. We found the set of all points where a ball bounced off the outfield wall, and solved for the 4-degree polynomial that best approximated those points with `np.polyfit`.

Fielder Interactions (in-depth)

A key issue was handling situations in which multiple outfielders could make the catch. Initially, our dataset contained one observation per fielder at each timestamp, resulting in three rows for each play. While this considered every possible fielder (giving us 3 times the data, especially for the less frequent class of missed catches), it created an imbalance for making the model: nearly two-thirds of all prediction points are for players not actively trying to reach the ball.

This led us to align with Statcast's methodology: we assigned the responsibility to the outfielder closest to the ball's final landing location. This resulted in reduced duplicated observations and created a better target. While this significantly decreased the number of missed catches to train, the quality of the training data was improved and narrowed to our specific use-case.

Direction Handling

We gave our model the sine and cosine of the angle from the player to the ball at each timestamp, in the same coordinate system used for plotting (90 degrees points to center field, 180 and 0/360 degrees point left and right, and 270 degrees points towards the plate. We used sine and cosine to represent angles in their vertical and horizontal components, because angles are circular features (0 and 360 degrees are the same), and with a limited-depth tree it was better to use two linear features (sine and cosine) which contain the same information.

For example, in a decision tree to check if an angle is in the 30 degree arc between 345 degrees and 15 degrees, one would have to build a tree like:

```

Angle > 345 degrees -> X
|___ If not then:
    |___ Angle < 15 degrees -> X
    |___ Else -> not X
  
```

With sin and cosine features, that decision is just
 Cosine > 0.9659 -> X

Statcast’s catch probability takes direction into account but only to determine whether the player must turn around to catch a ball in the 30 to 150 degree range (depicted in Figure D2)

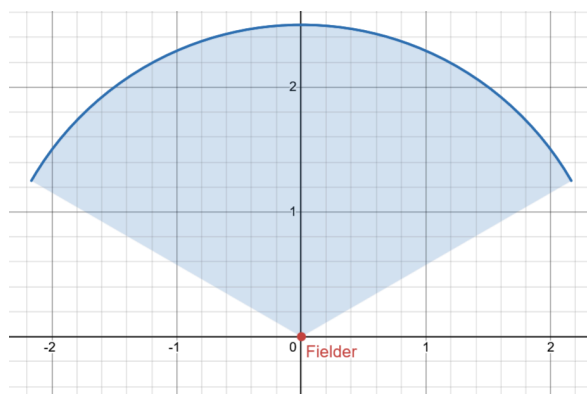


Figure D2: 30-150 degrees on the unit circle

Instead of using angle as a feature in their training process, they bump their required speed feature up by 1 ft/s (the same amount for all plays in that range) to account for the split second needed to turn around before they speed up.

Using angles with respect to the coordinate system we used, the assumption that the ball is at an angle between 30 and 150 degrees from the player will always produce a harder catch does not hold when predicting every timestamp of the play, as fielders can change direction and be sprinting directly towards e.g. center field. Because of this, we opted to give our model full information about a player’s angle to the ball, and let it figure out for itself in what cases direction was meaningful. There is also likely *some* nuance in whether a player having to range to his left or right affects catch probability, and we wanted to include this to make our visual circles more interesting.

If we knew what direction the fielder was facing at all times, we could measure angles relative to that direction at every timestamp and switch to “backward or not” like Statcast does. Then our noisy concentric circles would likely appear more cardioid-shaped (Figure D3), with all regions behind the direction the fielder is facing more difficult to make a catch in. As it stands, our concentric circles sometimes show cardioid-like patterns but often aren’t distinguishable from circles (Figure D4)

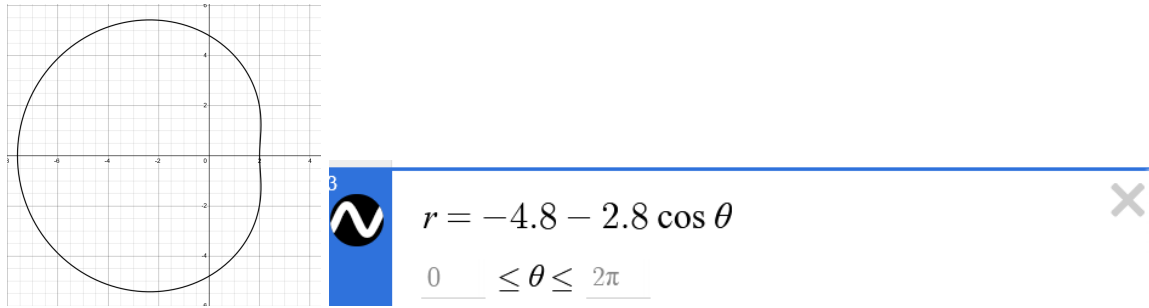


Figure D3: Example Cardioid, which we would expect a catch probability cloud with perfect knowledge to look like

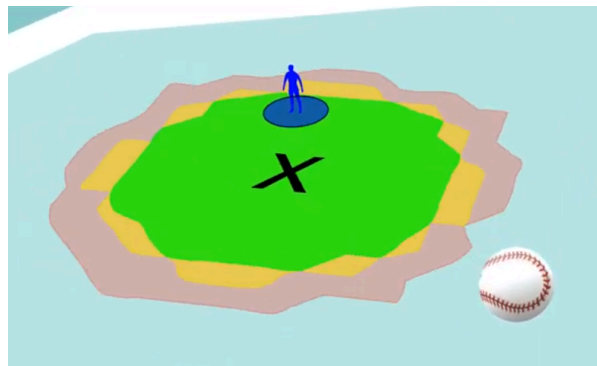


Figure D4: Differences from perfect circles produced by our model don't match the theoretical cardioid shape

Appendix E: Animation Technical Details

One of the best parts of Manim, and this animation extension, is that it can reasonably power a large variety of other plays and visualizations that occur on the baseball field. As such, we refer to it as an extension to the Manim library, with a flexible generation of a 1:1 baseball field and player movement (to the extent quality data allows) for a variety of animation purposes. As discussed below, the extension could be used to power point-of-view animations of batters, pitchers, fielders, or baserunners, complete with dynamic camera movement and ball tracking. We view this as just the beginning of the extension's capabilities, and are excited to implement additional functionality in the future.

Animation Timeline Creation:

To keep animation occurrences aligned with our data, the animation functions off of one large timeline, where each timestamp has location data or relevant points for the object at the given time. Each dynamic object in the animation was assigned a visualization ID to be used in their updater code (see next section), linked into the timeline through the relevant column sharing the same ID. To create movement over time, we initialize a Manim ValueTracker with a

value of 0. The animation runs by “animating” the value of the ValueTracker from 0 to a value equal to the number of rows in the timeline, with the speed controlled by a run time parameter. This ValueTracker is defined as the “row_tracker” because it essentially acts as a pointer to singular rows in the timeline, sliding down over the course of the animation. By flooring the value of row_tracker, the pointer always remains at the current relevant row without jumping into the future by rounding up. However, the intermediate values between rows can be used to power movement interpolation between known positions, improving the smoothness of object motion.

Object Movement Through Updaters:

With the timeline created, every dynamic part of the animation is given a defined updater function, either unique to it or shared between objects depending on its specific needs. Updaters allow us to translate the previously mentioned “row_tracker” values into actual movement. At each frame of the animation, the object fetches the necessary information from our timeline dataframe by combining the current row of the animation with the sharing the object’s visualization ID. With this information, the object updates its location (or shape in the case of our probability clouds) to keep in line with the real-world data. These updaters not only power general movement, but enable the small-but-important aspects of the visual like the probability clouds fading in as the ball is hit, the “Miss!” text only appearing after the ball bounces, or the color interpolation of the catch probability bar in the top left corner, which all improve visual coherence.

```
def updateObject(mobject):
    catch_timestamp = catch_probability_df['catch_timestamp'].iloc[0]

    animation_frame = row_tracker.get_value()
    current_row = min(int(row_tracker.get_value()), total_rows - 1)
    frac = animation_frame - current_row
    visualization_id = mobject.vis_id
    if (current_row + 1 < total_rows and
        timeline.iloc[current_row]['timestamp'] <= catch_timestamp):
        cell_value = timeline.iloc[current_row][visualization_id]
        if isinstance(cell_value, (tuple, list)):
            last_x = timeline.iloc[current_row][visualization_id][0]
            last_y = timeline.iloc[current_row][visualization_id][1]
            last_z = timeline.iloc[current_row][visualization_id][2]

            next_x = timeline.iloc[current_row + 1][visualization_id][0]
            next_y = timeline.iloc[current_row + 1][visualization_id][1]
            next_z = timeline.iloc[current_row + 1][visualization_id][2]

            last_pos = grid.c2p(last_x, last_y, last_z)
            next_pos = grid.c2p(next_x, next_y, next_z)

            current_pos = interpolate(last_pos, next_pos, frac)

            mobject.move_to(current_pos)
    if visualization_id == 'ball' and timeline.iloc[current_row]['timestamp'] > catch_timestamp:
        mobject.time_passed += 1

        progress = mobject.time_passed / 10
        new_opacity = max(1.0 - progress, 0.0)

        mobject.set_opacity(new_opacity)
    else:
        pass
```

Figure E1: One of the animation’s updater functions, this one for the ball and player movement

To further refine our movement, interpolation was added to fill in gaps between known locations and allow for a higher, smoother frame rate. Since the value tracker produces non-integer numbers we previously floored down into the relevant row, we can calculate the progress a player should have made between his last known position and next known position by subtracting the current “frame” value from the floored row value. Luckily, Manim provides a built-in interpolation function that uses this fractional progress value to calculate the intermediate points. While interpolation can produce confusing effects when handling large changes, the granularity of the data means it is only ever creating small shifts between known values.

Dynamic camera movements with Euler Angle Interpolation:

One of the most technically difficult aspects of this animation was creating dynamic camera work that really makes the animation feel like a video game¹³. This camera work is also powered by interpolation of the Euler angles (ManimGL uses the theta, phi, gamma ZXX system) to control the camera angle while it follows the center of the ball through space. This creates smooth movement regardless of the play in question, and opens the door to even more interesting camera work. In a previous iteration of the animation, we even designed code to dynamically track the ball from the perspective of the catching player, but this system didn’t allow us to visualize the catch probability clouds and was removed.

```
def camera_ball_flight_tracking_updater(camera_frame):
    hit_timestamp = catch_probability_df['hit_timestamp'].iloc[0]
    catch_timestamp = catch_probability_df['catch_timestamp'].iloc[0]

    hit_row = timeline.index[timeline['timestamp'] == hit_timestamp][0]
    catch_row = timeline.index[timeline['timestamp'] == catch_timestamp][0]

    hangtime_by_row = catch_row - hit_row

    animation_frame = row_tracker.get_value()

    flight_progress = (animation_frame - hit_row) / hangtime_by_row
    flight_progress = max(0.0, min(1.0, flight_progress))
    flight_progress = smooth(flight_progress)

    current_row = min(int(row_tracker.get_value()), total_rows - 1)
    ball_pos = baseball_tex.get_center()

    final_camera_angle = (0*DEGREES, 40*DEGREES, 0*DEGREES)
    final_height = 1

    if (timeline.iloc[current_row]['timestamp'] > hit_timestamp and
        timeline.iloc[current_row]['timestamp'] <= catch_timestamp):

        interpolated_theta = interpolate(starting_camera_angle[0], final_camera_angle[0], flight_progress)
        interpolated_phi = interpolate(starting_camera_angle[1], final_camera_angle[1], flight_progress)
        interpolated_gamma = interpolate(starting_camera_angle[2], final_camera_angle[2], flight_progress)

        interpolated_height = interpolate(starting_camera_height, final_height, flight_progress)

        camera_frame.set_height(interpolated_height)
        camera_frame.move_to(ball_pos)
        camera_frame.set_euler_angles(interpolated_theta, interpolated_phi, interpolated_gamma)
```

Figure E2: The code that enables our camerawork

¹³ The camera movement was inspired by *Backyard Baseball* (2004question)

```

def camera_ball_tracking_updater(camera_frame, dt):
    v = baseball_tex.get_center() - catching_player.get_center()
    theta_angle = math.atan2(v[1], v[0]) - (math.pi / 2)
    horizontal_dist = math.sqrt(v[0]**2 + v[1]**2)
    elevation_angle = math.atan2(v[2], horizontal_dist)
    phi_angle = (math.pi / 2) + elevation_angle
    gamma_angle = 0
    if horizontal_dist > 1:
        frame.move_to(catching_player.get_center())
        frame.align_to(catching_player, OUT)
        frame.set_height(0.001)
        frame.set_euler_angles(theta_angle, phi_angle, gamma_angle)
    else:
        frame.move_to(catching_player.get_center())
        frame.align_to(catching_player, OUT)
        frame.set_height(0.1)
        frame.set_euler_angles(theta_angle, phi_angle, gamma_angle)

```

Figure E3: Unused updater to track the ball from the fielder POV

Successes and Failures in Cloud Shape Generation

One of the most difficult challenges in the animation was creating meaningful but readable cloud shapes. At a high level, the boundary points were calculated as discussed previously, and then “side-filled” to fill in segments with no valid points with the previous point. This was a requirement for the Manim engine to handle the transitions between cloud shapes by ensuring each point always has a partner point at the same index in the next timestamp to move to. This required behavior did produce undesirable drawbacks, as the engine generates the lines between points even when points are on top of each other. This creates sharp corners when smooth curves are desired and intended. While Manim supports a function called “set_points_smoothly” that constructs smooth bezier curves between points, these curves fail when lines are generated between equivalent points. However, this design does allow the point locations to be interpolated across frames, creating “between-frame” smoothness despite struggles with “between-point” smoothness.

Constructing The Field’s Walls and Warning Track

Developing the walls of the field was integral to properly visualizing the foul territory of the field, an area where spectacular catches are still likely to appear. The foundation of the wall comes from Andrew Fox’s 2015 Fangraphs article¹⁴, which provides detailed equations to construct polar points for each MLB field’s outfield wall. These were then converted into Cartesian coordinates manually to create an array of points. Feeding these points into a ParametricSurface object in Manim with a custom function produced a wall with a Z-axis from the 2D points.

¹⁴ *Complete Outfield Dimensions* by Andrew Fox (2015). See Citations for a full citation.

To expand this foundation from the outfield wall to the entire perimeter of the stadium, we turned to Google Earth measurements. By manually measuring and recording the distance and angle from home plate to various points along the field's perimeter, we were able to construct additional polar coordinates. However, Google Earth measures angles by “heading”, which is relative to true north. As such, these measurements had to be manually altered because these heading values increased in a counterclockwise direction and true north was equivalent to 145.02 degrees relative to home plate.

```
def heading_to_theta(heading_dict):
    updated_dict = {}
    for heading, r in heading_dict.items():
        fixed_theta = abs( (heading + 34.98 + 180) % 360 - 360 )
        updated_dict[round(fixed_theta, 1)] = r

    return updated_dict
```

Figure E4: Our conversion from heading to home plate relative degrees

Armed with these additional points, we construct the rest of the field by linking together these points with Manim. Given more time and additional accurate measurements, the field perimeter could be entirely converted into a series of equations like the original Andrew Fox post. However, initial implementations of such a system lead to asymptotic behavior between closely related points and a confusing visual. Access to detailed field modeling that individual clubs likely have would improve this visualization further.